www.ietdl.org

Published in IET Software
Received on 26th October 2009
Revised on 2nd March 2011
doi: 10.1049/iet-sen.2009.0083

ISSN 1751-8806

Study on the relevance of the warnings reported by Java bug-finding tools

J.E.M. Araújo[2] *S. Souza*[1] *M.T. Valente*[2]

[1]*Institute of Informatics, PUC Minas, Brazil*
[2]*Department of Computer Science, UFMG, Brazil*
E-mail: mtvalente@gmail.com

Abstract: Several bug-finding tools have been proposed to detect software defects by means of static analysis techniques. However, there is still no consensus on the effective role that such tools should play in software development. Particularly, there is still no concluding answer to the following question usually formulated by software developers and software quality managers: how relevant are the warnings reported by bug finding tools? The authors first report an in-depth study involving the application of two bug-finding tools (FindBugs and PMD) in five stable versions of the Eclipse platform. Next, in order to check whether the initial conclusions are supported by other systems, the authors describe an extended case study with 12 systems. In the end, it has been concluded that rates of relevance superior to 50% can be achieved when FindBugs is configured in a proper way. On the other hand, in the best scenario considered in the research, only 10% of the warnings reported by PMD have been classified as relevant.

1 Introduction

Recently, several bug-finding tools have been proposed in order to detect defects in software by means of static analysis techniques [1–4]. Such tools look for code idioms denoting inconsistent code or violations in programming practices. By relying on such idioms (aka bug patterns), the tools can trigger warnings, for example, due to division by zero, overflow in arrays, uncaught exceptions, null pointer dereference, improper use of variables, synchronisation pitfalls, security vulnerabilities and so on. In summary, bug-finding tools contribute to extend and improve the warning messages typically generated by compilers. They can also be used to enforce coding style guidelines, such as indentation and naming conventions. Among such tools, we can mention Lint [5, 6] and PREfix/PREfast [7] (for programs in C/C++), FindBugs [8] and PMD [9] (for programs in Java) and FxCop [10] (for programs in C#).

Despite the increasing interest in bug-finding tools – both academically and by professional software developers – there is still no consensus on the effective role that such tools should play in software development [11–14]. Particularly, there is still no concluding answer to the following question usually formulated by software developers and software quality managers: how relevant are the warnings reported by bug-finding tools?

To contribute to an objective answer to this question, we first report in this paper an experience involving the application of two bug-finding tools in different versions of the Eclipse platform. The following tools have been used: FindBugs [8] and PMD [9]. These tools are chosen because they are probably the open-source tools, which are most popular nowadays for static analysing Java systems. In this first experience, both tools have been retrospectively applied in order to evaluate the lifetime of the warnings of five versions of the Eclipse platform. Two lessons have been learned in this first experience. First, developers should configure FindBugs to just report warnings from categories that make sense in their systems (e.g. warnings from the correctness category). In this way, the number of non-relevant warnings – or false positives – reported by the tool is reduced by a considerable amount. In our case study, proceeding in this way we have achieved relevance rates superior to 50% in some of the evaluated scenarios. Second, PMD generates too many warnings to review all of them manually. Moreover, the rate of relevance is not significant. For example, in the scenarios considered in our Eclipse case study, at most 10% of the reported warnings have been classified as relevant.

Next, we have replicated the proposed study to 12 open-source Java systems. In this extended case study, we have considered only warnings reported by the FindBugs tool (since the PMD results in the first case study were not encouraging). In five systems (Struts, JEdit, Pdfsam, Jython and Tomcat), we have observed warning lifetimes similar to the Eclipse case study. On the other hand, in seven systems we have observed that even the warnings from the correctness category have not been removed. In this case, we have contacted the system developers asking them about the relevance of such warnings. In three systems (JabRef, Jetty and ArgoUML), the developers have confirmed that the warnings are relevant, despite they have remained undetected for almost one year in the code. In such cases, the developers have arranged to fix the warnings in the next release of the systems. Finally, in four systems

(FreeMind, Jung, JGraph and Xerces), the developers have classified the warnings as false positives.

The structure of the rest of this paper is as follows. Section 2 provides background and related work information. Section 3 describes the methodology followed in our experiment, including information about the configuration of the bug-finding tools and the strategy we have used to calculate the lifetime of the reported warnings. Sections 4 and 5 present the Eclipse and the extended case study, respectively. Section 6 presents the limitations and the threats to the validity of the presented results and findings. Finally, Section 7 concludes the paper and outlines future research.

2 Background

This section provides an overview of bug-finding tools (Section 2.1) and describes related work about the empirical evaluation of such tools (Section 2.2).

2.1 Bug-finding tools

The focus of this paper is on bug-finding tools based on automated static analysis techniques. In this section, we present the FindBugs and PMD tools, used in the experience described in this paper.

FindBugs: FindBugs is an open-source tool that can detect more than 360 bug patterns by analysing Java bytecode [1, 8]. The detected bug patterns are classified into categories (such as correctness, performance, malicious code, bad practice etc.). FindBugs also assigns to each bug pattern a high, medium or low priority, according to heuristics defined by each pattern's implementation. The tool can be extended by defining new bug detectors, which are written directly in Java most of the time using the visitor design pattern to transverse the target program structure. In order to match bug patterns, detectors can also rely on intraprocedural control and data flow analysis.

Fig. 1 shows an example of a buggy code detected by FindBugs in the Eclipse source code (version 3.0). Fig. 2 shows the warning message generated by FindBugs after analysing this code fragment. Since Java evaluates Boolean expressions using short-circuit, this warning indicates that `c.isDisposed()` (line 131) will only be evaluated when the first sub-expression evaluates to `true`, therefore leading to a `NullPointerException`. In fact, Eclipse developers have fixed this bug in a later version by replacing the `&&` operator by a || operator.

```
127: private void setSmartButtonVisible(boolean visible) {
       .....
131:   if (c == null && c.isDisposed())
132:     return;
       .....
142: }
```

Fig. 1 *Buggy code detected by FindBugs in the Eclipse plataform (version 3.0)*

```
Null pointer dereference of c in setSmartButtonVisible(boolean)
Dereferenced at JavaStructureDiffViewer.java:[line 131]
In method internal.ui.compare.JavaStructureDiffViewer.setSmartButtonVisible(boolean)
Value loaded from c

Null pointer dereference
A null pointer is dereferenced here. This will lead to a NullPointerException
when the code is executed.
```

Fig. 2 *FindBugs warning example*

PMD: PMD is an open-source tool that supports a rich set of rules for detecting potential bugs and checking coding style [9]. For example, PMD supports rules for detecting unused code, code size related problems (e.g. excessive method length), questionable manipulation of `strings`, questionable manipulation of methods like `clone()` and `finalize ()` and so on. There are also rules for dealing with particular frameworks, such as Java Beans, JSP, JUnit, Android and so on. Different from FindBugs, PMD requires the source code of the target program, because constraint rules are defined over the abstract syntax tree (AST) of such programs. PMD rule sets can be extended by using either the visitor pattern or XPath expressions.

For example, suppose the abstract class presented in Fig. 3 (extracted from Eclipse 3.0). For this class, PMD generates the warning described in Fig. 4, recommending to declare as abstract the empty method `internalDispose`. In fact, developers have turned this method abstract in version 3.4.

2.2 Related work

Ayewah *et al.* [11] have evaluated the results generated by FindBugs in three large software projects, including Sun's JDK, Sun's Glassfish J2EE Server and portions of the Google's Java code base. For each project, they have manually classified the medium and high-priority warnings from the correctness category detected by FindBugs in the following way: trivial, having some functional impact, and having substantial functional impact. For example, from the 379 warnings generated when evaluating Sun's JDK, only 38 warnings have been classified as having substantial functional impact. However, as acknowledged in the paper, the proposed categorisation is open to interpretation. Particularly, Sun's

```
 62: abstract class AbstractInfoView extends ...{
       ...
422:   protected void internalDispose() {
423:   }
       ...
496: }
```

Fig. 3 *Example of class having a warning generated by PMD (Eclipse 3.0)*

```
<violation
 beginline="422" endline="423"
 begincolumn="19" endcolumn="9"
 rule="EmptyMethodInAbstractClassShouldBeAbstract"
 ruleset="Design Rules"
 package="org.eclipse.jdt.internal.ui.infoviews"
 class="AbstractInfoView"
 externalInfoUrl="http://pmd.sourceforge.net/rules/...."
 priority="1">

 An empty method in an abstract class should be abstract instead

</violation>
```

Fig. 4 *PMD warning example (for the class showed in Fig. 3)*

JDK project managers could have different opinion about the proposed classification, based on their experience with the system, including the importance of existing modules, the skills of the developers involved in the implementation of each module and so on.

Zheng *et al.* [14] have followed the goal question metric (GQM) paradigm to determine 'whether automated static analysis can help an organisation to economically improve the quality of software products'. To fulfil this goal, they have evaluated three large-scale C++ projects from an industrial partner, Nortel Networks, using three commercial bug-finding tools: Gimpel's FlexeLint [http://www.gimpel.com/html/flex.htm], Reasoning's Illuma [http://www.reasoning.com] and Klockwork's inForce and GateKeeper [http://www.klockwork.com]. However, it is not clear whether their results can be extrapolated to type-safe languages (such as Java). Furthermore, since their industrial partner relies on an external prescreening service, their experience has considered only the true positives that survived this service.

Wagner *et al.* [15] have applied the same bug detectors considered in our experience – FindBugs and PMD – in two industrial case studies. The ultimate goal was to determine whether these tools are effective to detect field defects, that is, defects that later would be reported by end-users. Particularly, they could not find a single warning generated by FindBugs and PMD that could be related to a field defect. The reason was that most field defects are due to logical faults (e.g. calls to wrong application program interface methods). Despite this fact, we believe it is also important to evaluate whether the warnings generated by bug-finding tools denote violations in recommended programming practices. Such violations could not contribute to incorrect program behaviour, but can lead for example to code that it is difficult to understand and evolve.

Kim and Ernst [12] have proposed a prioritisation algorithm for the warnings issued by bug-finding tools. To motivate their algorithm, they have evaluated the precision of the warnings generated by three tools (FindBugs, PMD and JLint [http://jlint.sourceforge.net]) when applied to three medium-sized programs. They define precision by the following ratio: (#warnings on bug-related lines)/(#warnings issued by the tool). A bug-related line is a line modified in a bug fix change, that is, a change performed in the system to fix bugs or other problems reported by end-users. Therefore similar to the work of Wagner *et al.* [15], their definition for precision does not consider warnings removed by developers in order to improve the internal quality of the code.

In a previous work, Kim and Ernst have computed the lifetime of the warnings in two open-source projects (Columba and jEdit). However, they acknowledge that their experiment was subject to ambiguities, since they measure lifetime at the file-level only. More specifically, they computed lifetime as the period between the first and the last appearance of a bug category in a given file. Therefore, when there are multiple warnings of the same category in a given file, their results do not consider single warning removals. Moreover, their results could have been influenced by refactorings such as package splitting or moving a class/method to another file.

Rutar *et al.* [16] have compared five Java-based bug-finding tools: Bandera, ESC, FindBugs, JLint and PMD. They have discussed the techniques that each of the tools is based on and finally they have proposed a meta-tool to combine the results of such bug detectors.

Wagner *et al.* [17] have compared three bug-finding tools (FindBugs, PMD and QJ Pro) with reviews and tests. Basically, they have concluded the following: (a) bug-finding tools detect a subset of the defects detected by reviews (e.g. only reviews can detect defects such as logical faults or wrong results from functions); (b) dynamic tests can find completely different defects than bug-finding tools (e.g. logical defects that are only visible when executing the software).

Nagappan and Ball [18] have proposed an empirical approach for the early prediction of pre-release defect density based on the warnings generated by static analysis tools. Using as case study the Windows Server 2003 project, they have concluded that there is a strong positive correlation between the static analysis defect density and the pre-release defect density determined by testing.

3 Methodology

In this section, we provide information about the bug-finding tools evaluated in our experiment (FindBugs and PMD) and the methodology we have used to compute the lifetime of the reported warnings.

Bug-finding tools: We have used FindBugs, version 1.3.6. The tool has been executed in command line mode, with the `-high` option enabled. This option defines that only high-priority warnings should be reported. High-priority warnings include, for example, bug patterns from the correctness category (e.g. dereferencing a null reference) but also bugs classified as malicious code (e.g. writing to a static field in a non-static method) or bad practice (e.g. overriding the `equals` but not the `hashCode` method).

We have also used PMD, version 4.2.5. PMD has been configured to just report bugs having the maximal priority (command line option `-minimumpriority = 1`). Furthermore, we have decided to discard warnings from the following rule sets: BraceRules, Import Statement, Code Size and Naming Rules. Basically, warnings in such rule sets represent violations in coding style or naming conventions. Therefore, owing to the purpose of our study, we have deliberately classified such warnings as non-relevant.

Calculating warning's lifetime: To calculate the lifetime of the warnings issued by both bug-finding tools, we have implemented a Perl script. This script receives as input the directories where the target systems have been installed. It then runs both FindBugs and PMD over such versions. Next, the output files with the warnings generated by each tool are processed, in order to generate a new file containing a signature for each warning. A warning signature is a quadruple with the following information: an abbreviation for the warning description and the location of the warning in the source code, including package, class and method. For example, the signature for the warning presented in Fig. 1 is the following quadruple: (`NP_ALWAYS_NULL, internal.ui.compare, JavaStructureDiffViewer, setSmartButtonVisible`).

Finally, the signature files are processed, in order to calculate the lifetime of the reported warnings. A warning's life is defined by the pair (v_p, v_q), where v_p denotes the version where the warning has been issued for the first time and v_q denotes the version where the warning has been removed. If the warning has not been removed until the last version considered in the experiment, then $v_q = \infty$. Therefore, assuming $v_q \neq \infty$, a warning's lifetime is defined by `date` (v_q) – `date` (v_p), where `date` (v) is the release date of version v.

doi: 10.1049/iet-sen.2009.0083

Evaluated warnings: For a given version, we have only considered warnings that are located in methods, classes and packages that have not been renamed in the further versions considered in the experiment. As mentioned before, a warning signature contains the name of the method, class and package where the warning has been detected. In case any of such elements are renamed, our algorithm for measuring lifetimes will assume that the warning has been removed and that a new warning has been inserted in the renamed element. For example, suppose an experiment including versions 3.1–3.4 from a given system. Suppose also a warning located in method m, class c, and package p from version 3.1. In this case, we will only consider this warnings if the components (m, c, p) also exist in versions 3.2, 3.3 and 3.4.

A warning could have been removed due to a major refactoring in the code of the method where it has been detected for the first time. For example, suppose the buggy code from Fig. 1. In this code, we will consider the warning removed by a change that just replaces the `&&` operator by the || operator. However, we will also consider the warning removed by a change that deletes the whole `if` statement (line 131). We have considered warning removals due to general refactorings for two reasons. First, they have no influence in case we conclude that the warnings are not relevant (i.e. if all the removals are not enough to denote relevance, so are just removals due to precise changes). Second, this approach is commonly adopted in studies that evaluate bug lifetimes [12, 19].

4 Eclipse case study

4.1 Experiment setup

We have evaluated five Eclipse versions, having number 3.x, where $0 \leq x \leq 4$. Table 1 provides detailed information about the evaluated versions, including their release date and size in terms of lines of code. The first evaluated version has been released on June, 2004 and the last evaluated version has been released on June, 2008. The first evaluated version has a little more than one million lines of code (MLOC); the last version has more than 1.7 MLOC. Table 1 also shows the time required by FindBugs and PMD to analyse each of the considered versions. The presented times have been measured in a AMD Turion x2 64, 2.0 GHz CPU, with 2 GB RAM and operating system Ubuntu 9.04. As we can observe, the time to execute FindBugs has ranged from 28 to 48 min. On the other hand, the time to execute PMD has ranged from 12 to 20 min. This difference reflects the fact that the analysis performed by FindBugs is more complex, requiring for some warnings control and data flow intraprocedural analysis.

Table 1 Eclipse versions, including their size (MLOC = million lines of code) and the execution time for the FindBugs and PMD tools (in minutes)

Version	Date	MLOC	FB time	PMD time
3.0	June, 2004	1.004	28	12
3.1	June, 2005	1.210	31	13
3.2	June, 2006	1.440	39	14
3.3	June, 2007	1.585	42	16
3.4	June, 2008	1.771	48	20

4.2 FindBugs results

Evaluated warnings: We have evaluated only warnings located in packages internal to the Eclipse platform (i.e. packages `org.eclipse.*`). Particularly, since FindBugs works at the bytecode level, it can report warnings located in external libraries (e.g. `jar` files). However, such warnings have been ignored in our experiment since they are not detected by PMD.

For the Eclipse versions considered in the study, Table 2 presents the total number of warnings reported by FindBugs and the number of warnings considered in our study. As can be observed, most of the ignored warnings are due to their location in external libraries (column B) than to renamed program elements (column C). Table 2 also shows a slightly reduction in the warnings density from version 3.0 (0.84 warnings/KLOC) to version 3.4 (0.63 warnings/KLOC).

Warnings' lifetime: Table 3 summarises the results from running FindBugs over the five versions considered in the experiment. The table shows the total number of warnings considered in each evaluated Eclipse version. It also details how many warnings are new and how many warnings have been propagated from previous versions. For example, in version 3.2, FindBugs has reported a total of 714 warnings, including 376 warnings that have been detected for the first time in version 3.0, 184 warnings detected for the first time in version 3.1 and 154 warnings that are new, that is, warnings that have been reported for the first time in version 3.2.

By analysing Table 3, we can also track the lifetime of the warnings reported in any of the evaluated versions. For example, from the 503 warnings reported for the first time in version 3.0, 394 warnings have been reported again in version 3.1, 376 warnings have been reissued in version 3.2 and so on. In summary, from the 503 warnings detected initially in version 3.0, 350 warnings have not been removed in the last evaluated version (around 70%).

Table 2 Total number of warnings reported by FindBugs (column A), warnings per KLOC, total number of warnings in external libraries (column B), total number of warnings in renamed program elements (column C) and total number of warnings considered in the experiment (column D)

Version	Total (A)	A/KLOC	External (B)	Renamed (C)	D = A − B − C	D/A
3.0	848	0.84	234	111	503	59%
3.1	887	0.73	231	59	597	67%
3.2	1043	0.72	276	53	714	68%
3.3	1162	0.73	336	24	802	69%
3.4	1111	0.63	142	0	969	87%

Table 3 Warning's lifetime as reported by FindBugs

Versions	#Warnings				
	3.0	3.1	3.2	3.3	3.4
3.0	503	394	376	355	350
3.1	–	203	184	175	177
3.2	–	–	154	142	130
3.3	–	–	–	130	122
3.4	–	–	–	–	190
Total	503	597	714	802	969

Table 4 Relevant warnings (%), as reported by FindBugs

Version	Lifetime's threshold		
	12 months	24 months	36 months
3.0	21.7	25.2	29.4
3.1	9.4	13.8	12.8
3.2	7.8	15.6	–
3.3	6.2	–	–

Relevant warnings: In this first case study, we consider that a warning is relevant when its lifetime is inferior than a time threshold t. Table 4 shows the percentage of relevant warnings for the following values of t: 12, 24 and 36 months. For version 3.2, the threshold of 36 months exceeds the duration of the experiment, and therefore has not been calculated. Similar observation holds for 24 and 36 months in version 3.3.

As presented in Table 4, the percentage of relevant warnings has ranged from 6.2% (version 3.3, $t = 12$ months) to 29.4% (version 3.0, $t = 36$ months). In summary, in the best evaluated scenario, Eclipse's developers have removed only 29.4% of the warnings reported by FindBugs.

Most common warnings: In order to understand why developers – after three years – have not removed most of the reported warnings, Table 5 presents the four most common warnings reported by FindBugs when applied to Eclipse 3.0.

The warnings included in Table 5 – representing more than 71% from the total number of reported warnings – are briefly described next:

- MS_SHOULD_BE_FINAL: This warning, which represents 38% from the total number of warning occurrences, is reported when FindBugs detects a static field not declared as final. FindBugs' developers have classified such declarations as a warning because mutable static fields could be changed by malicious code or by accident from another package. However, this warning category is only relevant in systems that access sensitive data. Certainly, this is not the case of development platforms, such as Eclipse.
- ST_WRITE_TO_STATIC_FROM_INSTANCE_METHOD: This warning, which counts for almost 19% from the total number of warning occurrences, is reported when a non-static method writes to a static field. FindBugs' developers claim that this kind of code is tricky to get correct if multiple instances are being manipulated. In fact, according to FindBugs categories, this warning is classified as dodgy code.
- HE_EQUALS_NO_HASHCODE: This warning, counting for more than 7% from the total number of warnings, is reported when a class overrides `equals(Object)`, but does not override `hashCode()`. Therefore the class may violate the invariant that equal objects must have equal hashcodes. This warning is classified as a bad practice.

Table 5 Most common warning categories (Eclipse 3.0)

Warning	Category	Occurrence	%
MS_SHOULD_BE_FINAL	malicious code	191	38.0
ST_WRITE_TO_STATIC_FROM_INSTANCE_METHOD	dodgy	95	18.9
HE_EQUALS_USE_HASHCODE	bad practice	38	7.5
MS_MUTABLE_ARRAY	malicious code	35	7.0
Total	–	359	71.4

- MS_MUTABLE_ARRAY: This warning is reported when a static final field that references an array can be accessed by malicious code or by accident from another package. In this way, by accessing this field, it is possible to freely modify the contents of the array. Once more, this represents a warning that is important only in systems that access sensitive data or systems with restrictive security requirements.

In summary, the mentioned warnings denote recommended programming practices (e.g. HE_EQUALS_NO_HASH CODE) or warnings that are relevant only in particular application domains (e.g. MS_SHOULD_BE_FINAL). It is also important to mention that (a) FindBugs has ranked the four mentioned warnings as high-priority; (b) none of the four mentioned warnings are from the correctness category.

Discussion: Based on the results from Table 4, we initially concluded that at least in our Eclipse case study most of the warnings reported by FindBugs are not relevant. On the other hand, the results from Table 5 show that FindBugs consider as high-priority warnings that denote suspicious, tricky or non-standard code and warnings that are important only in particular application domains. To better support this last conclusion, we have executed again FindBugs over the five Eclipse versions, but configured the tool to just report high-priority warnings from the correctness category. Table 6 reports the lifetime of the warnings in this category. As can be observed, correctness warnings represent from 10.8 to 12.1% from all warnings detected by FindBugs. Table 7 presents the percentage of relevant warnings in this category.

When comparing Tables 4 and 7, we observe a considerable increase in the percentage of relevant warnings. For example, in the best scenario from Table 4, 29.4% of the warnings have been classified as relevant. When restricting the analysis to just warnings in the correctness category, this percentage has increased to 64%, as presented in Table 7. On the other hand, in the worst scenario from Table 4, just 6.2%

Table 6 Warning's lifetimes (just correctness category), as reported by FindBugs

Versions	#Warnings				
	3.0	3.1	3.2	3.3	3.4
3.0	61	35	28	22	17
3.1	–	35	25	23	23
3.2	–	–	24	17	15
3.3	–	–	–	33	28
3.4	–	–	–	–	29
Total	61	70	77	95	112
% all categories	12.1	11.7	10.8	11.8	11.6

Table 7 Relevant correctness warnings (%), as reported by FindBugs

Version	Lifetime's threshold		
	12 months	24 months	36 months
3.0	42.6	54.1	64.0
3.1	28.6	34.3	34.3
3.2	29.2	37.5	–
3.3	15.2	–	–

doi: 10.1049/iet-sen.2009.0083

Table 8 Most common correctness warnings (Eclipse 3.0), as reported by FindBugs

Warning	Occurrence	%
NP_NULL_ON_SOME_PATH	14	23.0
RCN_REDUNDANT_NULLCHECK_ WOULD_HAVE_BEEN_A_NPE	7	11.5
NP_ALWAYS_NULL	6	9.8
RV_RETURN_VALUE_IGNORED	5	8.2
Total	32	52.5

of the warnings have been classified as relevant. When restricting the analysis to correctness warnings, this percentage has increased to 15.2%.

Table 8 presents the four most common warnings from the correctness category (as reported in Eclipse 3.0). In this table, the first three warnings are related to dereferencing a null pointer, which is clearly a bug in Java programs. Not surprisingly from the 32 warnings presented in this table, 27 warnings have been fixed in later versions of the Eclipse platform.

Conclusions: At least as demonstrated by this first case study, it is fundamental that developers configure FindBugs to report warnings from categories that make sense in their systems. In this way, the number of non-relevant warnings – or false positives – reported by the tool is reduced by a considerable amount. Proceeding in this way, we have achieved relevance rates superior to 50% in some of the evaluated scenarios.

4.3 PMD results

Evaluated warnings: For each Eclipse version, Table 9 presents the total number of warnings reported by PMD and the total number of warnings considered in our study. With PMD, we only discarded warnings in renamed program elements, since the tool does not analyse external JAR files.

Table 9 Total number of warnings reported by PMD (column A), warnings per KLOC, number of warnings in renamed program elements (column B) and number of warnings considered in the experiment (column C)

Version	Total (A)	A/KLOC	Renamed (B)	C = A − B	C/A
3.0	6176	6.15	2186	3990	65%
3.1	6961	5.75	2154	4807	69%
3.2	7820	5.43	2372	5448	70%
3.3	8326	5.25	2233	6093	73%
3.4	9129	5.15	2280	6849	75%

Table 10 Warning's lifetime as reported by PMD

Version	#Warnings				
	3.0	3.1	3.2	3.3	3.4
3.0	3990	3832	3688	3594	3582
3.1	–	975	918	889	877
3.2	–	–	842	806	790
3.3	–	–	–	804	784
3.4	–	–	–	–	816
Total	3990	4807	5448	6093	6849

Warnings' lifetime and relevance: Table 10 summarises the results from running PMD over the five versions of the Eclipse platform considered in the experiment. As we can observe in this table, PMD generates much more warnings than FindBugs. For example, while FindBugs has reported 969 warnings for Eclipse 3.4, PMD has reported 6849 warnings (an increase of 607%). However, PMD does not present an impressive rate of relevant warnings, as reported in Table 11. In the best scenario, only 10.1% from the reported warnings have been classified as relevant (Eclipse 3.1, $t = 36$ months).

Table 12 shows the number of occurrences for each warning reported by PMD when executed over Eclipse 3.0. The table also shows how many warnings have been removed in later versions of the system. Owing to our previous filtering of warnings related to naming and style conventions, PMD has reported only 11 types of warnings in this Eclipse version. From such warnings, ten warnings are from the following rule sets: design rules (e.g. *TooFewBranchesForASwitchStatement*), strict exception rules (e.g. *AvoidThrowingNullPointerException*) and controversial rules (e.g. *AvoidUsingShortType*). In fact, from the reported warnings, just *DoubleCheckedLocking*

Table 11 Relevant warnings (%), as reported by PMD

Version	Lifetime's threshold		
	12 months	24 months	36 months
3.0	4.0	7.6	9.9
3.1	5.8	8.8	10.1
3.2	4.3	6.2	–
3.3	2.5	–	–

Table 12 Warnings reported by PMD (Eclipse 3.0)

Warning	Rule set	Occurrence	%	Removed
EmptyMethodInAbstractClass	Design_Rules	1284	32.2	62
AvoidUsingShortType	Controversial_Rules	1232	30.9	151
ConstructorCallsOverridableMethod	Design_Rules	714	17.9	126
TooFewBranchesForASwitchStatement	Design_Rules	307	7.7	73
ReturnEmptyArrayRatherThanNull	Design_Rules	223	5.6	45
AvoidThrowingRawExceptionTypes	Strict_Exception_Rules	124	3.1	23
AvoidThrowingNullPointerException	Strict_Exception_Rules	99	2.5	4
EqualsNull	Design_Rules	6	0.1	1
DoubleCheckedLocking	Basic_Rules	1	0.0	1
Total		3990	100	486

could possibly generate a defect. This warning is reported when a variable assigned within a synchronised section is used outside of this section.

Conclusions: We have concluded that PMD is not an effective tool to report relevant warnings, according to our classification of relevance based on lifetime. The rate of relevant warnings reported by the tool has ranged from 2.5 to 10.1%. Furthermore, PMD generates much more warnings than FindBugs. Particularly, we believe that PMD is more effective to issue warnings related to stylistic violations, such as in indentation and naming conventions.

5 Extended case study

To check whether our initial findings apply to other systems, we have conducted an extended experiment involving 12 open-source systems. In this second study, we have made two important decisions: (a) we have limited the evaluation to the warnings reported by the FindBugs tool (since PMD has generated numerous false positives in the Eclipse study); (b) we have limited the evaluation to two versions of each target system, with a time interval close to one year between them. The systems – and their versions – considered in the study are presented in Table 13.

Table 14 presents the lifetime of the warnings in this second experiment – which have been calculated using the same methodology from the Eclipse study. The first column shows the number of warnings reported by FindBugs when executed over the initial version of the considered systems. From those warnings, the second column also shows how many warnings have been reported again in the final evaluated version. For example, five warnings have been reported for the initial version of the Struts2 system. From those warnings, only one has remained in the final version evaluated in the study. Finally, the table reports the relevance of the detected warnings (i.e. the percentage of warnings detected in the initial version and that have been removed in the final version). In only three systems we have achieved relevance rates close or superior to 40% (Struts2, Pdfsam and JEdit). For the remainder nine systems, the relevance rates have been inferior to 20%. Such numbers are compatible with the results described in the Eclipse case study. They reinforce our initial argument that the general-purpose, high-priority warnings reported by FindBugs are not relevant.

Table 14 High-priority (HP) warning's lifetime and relevance

	System	# HP warnings		Relevance, %
		Initial	Final	
1	Struts2	5	1	80.0
2	Pdfsam	5	3	40.0
3	JEdit	37	23	37.8
4	Jython	74	60	18.9
5	Tomcat	144	126	12.5
6	FreeMind	43	41	4.7
7	JabRef	149	148	0.7
8	Jung	15	15	0.0
9	JGraph	25	25	0.0
10	ArgoUML	42	42	0.0
11	Jetty	43	43	0.0
12	Xerces	48	48	0.0

In the next step, we have executed again FindBugs over the same systems, but with the tool configured to just report high-priority warnings from the correctness category. Table 15 summarises the results we have achieved in this second execution. In five systems, we have achieved relevance rates superior to 40%. However, in the remainder seven systems, the relevance rates have been equal to zero.

Table 15 High-priority and correctness (HP and CT) warning's lifetime and relevance

	System	# HP and CT warnings		Relevance (%) (lifetime)	Relevance (%) (developers)
		Initial	Final		
1	Struts2	2	0	100.0	–
2	JEdit	15	3	80.0	–
3	Pdfsam	3	1	66.7	–
4	Jython	10	4	60.0	–
5	Tomcat	24	14	41.7	–
6	JabRef	7	7	0.0	85.7
7	Jetty	4	4	0.0	75.0
8	ArgoUML	11	11	0.0	54.5
9	FreeMind	6	6	0.0	16.6
10	Jung	3	3	0.0	0.0
11	JGraph	2	2	0.0	0.0
12	Xerces	13	13	0.0	0.0

Table 13 Systems evaluated in the extended case study

	System	Initial version			Final version			
		Number	Date	KLOC	Number	Date	KLOC	Interval (days)
1	JEdit	4.0.3	06/2002	42.1	4.1	05/2003	46.9	343
2	Jung	1.7.1	10/2005	30.6	1.7.5	10/2006	31.8	363
3	FreeMind	0.9.0b17	05/2008	38.4	0.9.0RC4	05/2009	39.6	381
4	JGraph	5.10.2.0	11/2007	14.8	5.12.2.1	11/2008	15.0	350
5	ArgoUML	0.26	09/2008	135.4	0.28.1	08/2009	134.5	323
6	Tomcat	5.5.9	10/2005	116.9	6.0.0	10/2006	117.6	371
7	Struts2	2.0.14	11/2008	38.6	2.1.8.1	11/2009	75.1	354
8	Jetty	6.1.14	11/2008	63.1	6.1.22	11/2009	67.4	368
9	Xerces	2.7.1	7/2005	82.5	2.8.1	9/2006	84.0	414
10	Jython	2.5a3	09/2008	74.1	2.5.1rc3	09/2009	156.3	378
11	JabRef	2.5b	04/2009	55.0	2.6b3	03/2010	57.0	320
12	Pdfsam	1.1.2	04/2009	12.7	2.2.0	03/2010	19.9	358

doi: 10.1049/iet-sen.2009.0083

Therefore, since in more than 50% of the systems we have achieved results different from Eclipse, we decided to directly contact the system developers. We posted a message in the bug tracker of each system, with a brief description about the reported warnings. In this message, we also asked the developers to evaluate the importance of the warnings. As presented in the last column of Table 15, after the feedback from the developers, the warnings reported in three systems have been considered relevant (JabRef, Jetty and ArgoUML). In such cases, the developers have fixed the warnings in the source code.

On the other hand, in the case of four systems, the developers have classified the high-priority warnings from the correctness category as false positives (with the sole exception of one warning in FreeMind). For example, FreeMind's developers have not considered relevant a warning informing that some calls to `equals` would evaluate to false at runtime (in fact, they claimed this should not always be the case). Jung's developers have not agreed about warnings reporting an incorrect format argument in a `String.format` call. They argued that the resulting string is not a 'pretty string', but that the calls are still 'semantically valid'. For Jung, FindBugs has issued warnings about tests for equality to `Double.NaN` (i.e. code that checks to see if a double is equal to the special 'not a number (NaN)' value; however, because of its special semantics, no value is equal to NaN). Although classifying the warnings as 'fair enough', one of the Jung's developer said he is not too worried about them, since they have been reported in a legacy codebase. Finally, Xerces' developers have classified the warnings as irrelevant because they have been detected in dead code (i.e. code that is not called anymore).

Conclusions: The extended case study has contributed to strengthen our initial conclusions for Eclipse. Basically, the experience with 12 other systems has showed the importance of configuring the tool to report the most adequated warning's categories. When considering warnings from all categories in nine out of 12 systems, we have observed rates of relevance inferior to 20%. Since the results have been compatible with the Eclipse study (and to our personal evaluation of the reported warnings), we decided not to contact the developers about such general-purpose warnings. On the other hand, in five out of 12 systems we have observed warnings' lifetime similar to the Eclipse study after restricting the analysis to high-priority and correctness warnings. After contacting the developers, we have concluded about the relevance of the warnings reported in three more systems. Finally, in the systems where the developers have classified the warnings as not relevant, the arguments have not been technical (e.g. the result is still 'semantically valid', the developers have 'limited time to work on the project' etc.).

6 Threats to validity

In this section, we discuss potential threats to the validity of our study. As usual in empirical studies in software engineering, we have arranged possible threats in three categories: external validity, internal validity and construct validity [20].

External validity: This form of validity assesses the degree to which we can extend the results of a study to a wider population. Although we have evaluated more systems than in similar studies described in the literature, we acknowledge that our results may not be representative of all (open-source) software systems. Particularly, factors such as system's size and maturity and the experience of the community of developers can have a fundamental impact on the type of study reported on this paper. Finally, only two tools have been evaluated in the study. However, since FindBugs and PMD are the two most popular Java-based bug-finding tools, we believe that our conclusions are at least representative for tools available in this language. On the other hand, we stress that they cannot be generalised to memory-unsafe languages, like C and C++.

Internal validity: This form of validity assess whether the study findings are due to controlled variables or to unknown causes. In order to discard uncontrolled interference in the study, we have, for example, checked with Eclipse developers whether FindBugs or PMD are applied as part of their regular building process. The developers have confirmed to us that they do not use neither of the tools in the Eclipse development cycle.

Construct validity: This form of validity assesses the ability to draw statistically correct conclusions from the results produced by the experiments. In other words, the goal is to evaluate whether the proposed conclusions are not affected by misleading data. For example, as mentioned in Section 3, a warning signature is a quadruple with an abbreviation for the warning description and the warning location in the code, including package, class and method name. For this reason, to avoid misleading lifetime measures due to refactorings, for a given version, we have evaluated only warnings located in program elements present in all other versions considered in the study.

7 Conclusions

We have initially determined the lifetime of the warnings reported by two popular Java-based bug-finding tools – FindBugs and PMD – when executed over five stable releases of the Eclipse platform. Next, we have repeated this experiment to other 12 open-source systems. Our findings can be summarised in the following way:

- When using FindBugs, it is fundamental that developers configure the tool to just report warnings from categories that make sense in their systems. In this way, the number of non-relevant warnings – or false positives – reported by the tool is reduced by a considerable amount. In our case study, proceeding in this way we have achieved relevance rates superior to 50% in some of the evaluated Eclipse versions and in eight out of the 12 systems of the extended case study.
- PMD generates too many warnings to review all of them manually. Moreover, the rate of relevance is not significant. For example, in the scenarios considered in the Eclipse study, at most 10.1% of the reported warnings have been classified as relevant.

As future work, we have plans to expand our research in two directions: (a) including new subjected systems preferably from different domains (e.g. web-based systems and real-time systems) (b) including new bug-finding tools (e.g. tools for C/C++). We also have plans to design a meta-tool for finding bugs. This tool could be used for example to track, combine and correlate results from different bug-finding tools.

8 Acknowledgments

This research has been supported by grants from FAPEMIG, CAPES and CNPq.

9 References

1 Ayewah, N., Hovemeyer, D., Morgenthaler, J.D., Penix, J., Pugh, W.: 'Using static analysis to find bugs', *IEEE Softw.*, 2008, **25**, (5), pp. 22–29
2 Foster, J.S., Hicks, M.W., Pugh, W.: 'Improving software quality with static analysis'. 7th Workshop on Program Analysis for Software Tools and Engineering (PASTE), 2007, pp. 83–84
3 Deissenboeck, F., Juergens, E., Hummel, B., Wagner, S., Mas y Parareda, B., Pizka, M.: 'Tool support for continuous quality control', *IEEE Softw.*, 2008, **25**, (5), pp. 60–67
4 Louridas, P.: 'Static code analysis', *IEEE Softw.*, 2006, **23**, (4), pp. 58–61
5 Darwin, I.F.: 'Checking C programs with Lint' (O'Reilly, 1988)
6 Johnson, S.C.: 'Lint: A C program checker'. Technical report 65, Bell Laboratories, December 1977
7 Larus, J.R., Ball, T., Das, M., *et al.*: 'Righting software', *IEEE Softw.*, 2004, **21**, (3), pp. 92–100
8 Hovemeyer, D., Pugh, W.: 'Finding bugs is easy', *SIGPLAN Notices*, 2004, **39**, (12), pp. 92–106
9 Copeland, T.: 'PMD applied' (Centennial Books, 2005)
10 Microsoft Corporation: 'FxCop home page', available at http://msdn.microsoft.com/en-us/library/bb429476(VS.80).aspx
11 Ayewah, N., Pugh, W., Morgenthaler, J.D., Penix, J., Zhou, Y.: 'Evaluating static analysis defect warnings on production software'. 7th Workshop on Program Analysis for Software Tools and Engineering (PASTE), 2007, pp. 1–8
12 Kim, S., Ernst, M.D.: 'Which warnings should I fix first?'. 15th Int. Symp. on Foundations of Software Engineering (FSE), 2007, pp. 45–54
13 Kremenek, T., Engler, D.R.: 'Z-ranking: using statistical analysis to counter the impact of static analysis approximations'. 10th Int. Symp. on Static Analysis (SAS), 2003, (*LNCS*, **2694**), pp. 295–315
14 Zheng, J., Williams, L., Nagappan, N., Hudepohl, J.P., Vouk, M.A.: 'On the value of static analysis for fault detection in software', *IEEE Trans. Softw. Eng.*, 2006, **32**, (4), pp. 240–253
15 Wagner, S., Aichner, M., Wimmer, J., Schwalb, M.: 'An evaluation of two bug pattern tools for Java'. 1st Int. Conf. on Software Testing, Verification, and Validation (ICST), 2008, pp. 248–257
16 Rutar, N., Almazan, C.B., Foster, J.S.: 'A comparison of bug finding tools for Java'. 15th Int. Symp. on Software Reliability Engineering (ISSRE), 2004, pp. 245–256
17 Wagner, S., Jürjens, J., Koller, C., Trischberger, P.: 'Comparing bug finding tools with reviews and tests'. 17th Int. Conf. Testing of Communicating Systems (TestCom), 2005, (*LNCS*, **3502**), pp. 40–55
18 Nagappan, N., Ball, T.: 'Static analysis tools as early indicators of pre-release defect density'. 27th Int. Conf. on Software Engineering (ICSE), 2005, pp. 580–586
19 Kim, S., Ernst, M.D.: 'Prioritizing warning categories by analyzing software history'. 4th Int. Workshop on Mining Software Repositories (MSR), 2007, pp. 27–30
20 Perr, D.E., Porter, A.A., Votta, L.G.: 'A primer on empirical studies (tutorial)'. Tutorial Presented at 19th Int. Conf. on Software Engineering (ICSE), 1997, pp. 657–658

doi: 10.1049/iet-sen.2009.0083