Software Qual J
DOI 10.1007/s11219-011-9172-5

Static correspondence and correlation between field defects and warnings reported by a bug finding tool

Cesar Couto · João Eduardo Montandon · Christofer Silva · Marco Tulio Valente

Abstract Despite the interest and the increasing number of static analysis tools for detecting defects in software systems, there is still no consensus on the actual gains that such tools introduce in software development projects. Therefore, this article reports a study carried out to evaluate the degree of correspondence and correlation between post-release defects (i.e., field defects) and warnings issued by FindBugs, a bug finding tool widely used in Java systems. The study aimed to evaluate two types of relations: static correspondence (when warnings contribute to find the static program locations changed to remove field defects) and statistical correlation (when warnings serve as early indicators for future field defects). As a result, we have concluded that there is no static correspondence between field defects and warnings. However, statistical tests showed that there is a moderate level of correlation between warnings and such kinds of software defects.

Keywords Bug finding tools · Field defects · Software quality assurance tools

1 Introduction

There is a growing interest in static analysis techniques and tools for detecting bugs in software systems (Ayewah et al. 2008; Bessey et al. 2010; Foster et al. 2007; Louridas

C. Couto · J. E. Montandon · M. T. Valente (✉)
Department of Computer Science, UFMG, Belo Horizonte, Brazil
e-mail: mtov@dcc.ufmg.br

C. Couto
e-mail: cesarfmc@dcc.ufmg.br

J. E. Montandon
e-mail: joao.montandon@dcc.ufmg.br

C. Couto · C. Silva
Department of Computing, CEFET-MG, Belo Horizonte, Brazil

C. Silva
e-mail: christofer@dcc.ufmg.br

Published online: 14 December 2011

2006). Usually, these tools work by looking for violations in recommended programming practices, instead of checking whether a system meets its specification. As examples of the bugs detected by such tools, we can mention null pointer dereferences, improper use of synchronization primitives, division by zero, and overflow in arrays. In summary, such tools extend and improve the warning messages typically reported by compilers. Additionally, they can check coding style guidelines, including naming conventions and indentation patterns. Among the many bug finding (or bug pattern) tools that exist, we can mention Lint (Johnson 1977) and PREfix/PREfast (Larus et al. 2004) (for C/C++ programs), FindBugs (Hovemeyer and Pugh 2004) and PMD (Copeland 2005) (for Java programs) and FxCop (Microsoft Corporation http://msdn.microsoft.com/en-us/library/bb429476 (VS.80).aspx.) FxCop home page (for .NET programs).

Despite the increasing number of available bug finding tools, there is still no consensus on the effective power of these tools to detect post-release defects (also called field defects). More specifically, we still lack realistic exploratory studies that can help developers, maintainers and software quality managers to get clear answers for two central questions:

- **Question Q1 (Static correspondence):** Is there a correspondence between the static location of warnings and the program elements changed in order to remove field defects? In others words, are the warnings issued by static analysis tools located in the program elements changed by developers in order to remove field defects? If the answers to these questions are affirmative, we will consider in this article that there is a static correspondence between warnings and field defects. From a practical point of view, the proposed definition for static correspondence means that a developer designated to remove a field defect can take benefit from running a bug finding tool before starting this task. Furthermore, the proposed definition supports the implementation of an automatic procedure for evaluating the static correspondence between warnings and field defects, without requiring feedback from the developers in charge of the maintenance task.
- **Question Q2 (Correlation):** Is there a statistical relation between warnings and field defects? For example, the greater the number of warnings reported by a bug finding tool, the greater will be the number of field defects observed later in the system? If the answers to these questions are affirmative, we will consider in this article that there is a correlation between warnings and field defects. From a practical point of view, correlation means that a software project manager may consider the number of warnings as an early indicator of the quality of a system. More specifically, managers can establish that the new releases of a given system should have a warning density inferior than a certain threshold.

Initially, we assess in the article the static correspondence between warnings reported by FindBugs (Hovemeyer and Pugh 2004)—a widely used open source bug finding tool for Java—and field defects. More specifically, we have evaluated the usefulness of the warnings reported by FindBugs to detect and remove defects in three medium-to-large size systems: Rhino (a JavaScript interpreter with 31 KLOC that is developed as part of the Mozilla project), `ajc` (the most used AspectJ compiler, with around 63 KLOC), and Lucene (an information retrieval software library, with around 24 KLOC). Additionally, in a second experiment, we have assessed the statistical correlation between warnings generated by FindBugs and field defects reported for thirty systems maintained by the Apache Foundation, totaling almost one and a half

million lines of code. Our results show the existence of an important level of correlation between these two variables.

The remainder of this article is organized as follows. In Sect. 2, we describe the experiment designed to evaluate the existence of static correspondence between warnings and field defects. Section 3 describes the study conducted to evaluate the level of correlation between such variables. Section 4 summarizes the main lessons learned with the reported experiences. In Sect. 5, we document the main risks and limitations of the study described in this paper. Section 6 presents related work and Sect. 7 describes our conclusions.

2 Static correspondence

Suppose a field defect *d* reported to a system. Suppose that *C* denotes the program elements that must be changed in order to fix *d*. We consider that there is a static correspondence between a warning *w* reported by a bug finding tool and the defect *d* if *w* has been reported in one of the program elements in *C*. In other words, our assumption is that the causes of *d* are restricted to the program elements in *C* (since they represent the only elements changed in order to fix *d*). Therefore, warnings located in *C* have at least matched the defective program elements. However, two important observations should be made about this definition. First, it deliberately does not define the granularity of the program elements considered in the set *C*. Particularly, in the experiment described in this section we will only consider method level granularity. Second, it considers that there is a static correspondence between *w* and *d* even when *w* persists in *C* after *d* has been removed (i.e., in cases where developers have been able to fix *d* without removing *w*). However, our claim is that despite not been removed, such warnings are relevant because they at least represent a true alert for defective program elements (and therefore they can help maintainers to locate such elements when starting a bug fixing task).

Three systems—Rhino, `ajc`, and Lucene—have been used in this first experiment. For the first two systems, we have relied on the information available at the iBugs repository.[1] iBugs stores the source code before and after the correction of several defects reported by the users of the systems made available in the repository. Therefore, by comparing the provided source code (before and after a fixed bug), we can determine the methods at the aforementioned set *C*, i.e. the methods changed by the maintainers to fix a field defect *d*. For Lucene, we have relied on the information available at the Jira issue tracking and management system used by the projects hosted by the Apache Foundation.[2] Jira stores information about the source code files changed to fix defects reported by Apache's users. In addition, Jira provides the revision number of such files in the version-control system. Therefore, it was possible to discover the source code before and after the correction of the evaluated defects.

In iBugs, there are 32 issues (an issue may be a field defect, an improvement or new feature) reported to the Rhino interpreter. These issues have been reported via Bugzilla[3], the bug tracking system normally used by the systems of the Mozilla Foundation. Furthermore, iBugs provides information about 348 issues reported for the `ajc` compiler

[1] http://www.st.cs.uni-saarland.de/ibugs.

[2] https://issues.apache.org/jira/secure/IssueNavigator.jspa.

[3] http://www.bugzilla.org.

(such issues have been reported by 13 developers involved in the ajc implementation, also using Bugzilla). For Lucene, we have considered 90 post-release issues.

2.1 Data collection

In order to collect data to evaluate the degree of static correspondence between warnings and field defects, we have performed the following tasks:

2.1.1 Filtering the maintenance requests

For Rhino and ajc, we have carefully read and evaluated the free text of each maintenance request reported via Bugzilla. The aim was to distinguish between entries demanding corrective maintenance tasks (field defects) and entries that in fact are requests for adaptive, evolutive or perfective maintenance. As an example of the first case, we can mention the following bug reported to the ajc compiler: *there is not enough memory to compile an aspect developed by the user*. As an example of the second case, we can mention the following issue posted for the Rhino interpreter: *The system does not accept strings with more than 64 KB characters*. For Lucene, the process of filtering the maintenance requests was simpler, because Jira has a field that classifies the issues as *bugs, improvements*, and *new features*.

In our study, we have evaluated only entries associated with corrective maintenance tasks, because it makes no sense to expect a bug finding tool based on static analysis techniques to help on adaptive, evolutive or perfective maintenance tasks. For each evaluated system, Table 1 reports the number of entries classified as corrective maintenance and as the other maintenance types. As can be observed in this table, the percentage of requests classified as corrective has been 50% (for Rhino), 66% (for ajc), and 33% (for Lucene).

2.1.2 Calculating the static correspondence

First, we have downloaded the versions before and after each maintenance request we have classified as corrective. For Rhino and ajc, the source code has been retrieved directly from the iBugs repository. For Lucene, the ID of the SVN transaction responsible for fixing a given bug b is available in the Jira issue tracking system. Using this ID, we have accessed SVN to retrieve the source code of the version where b has been fixed. We have also retrieved from SVN the version of the system with an identifier equal to (ID-1), i.e., the version just before fixing bug b.

In a second step, we have compared the versions before and after fixing the evaluated bugs in order to find the defective methods. For this purpose, suppose that M_b and M_a are, respectively, the methods in the versions before and after fixing a given field defect d. In order to calculate M_b and M_a we have implemented a small parser for Java. From the Abstract Syntax Tree (AST) generated by this parser, it was possible to retrieve te

Table 1 Classification of the maintenance requests

Maintenance types	Rhino		ajc		Lucene	
	Qty	%	Qty	%	Qty	%
Corrective	16	50	231	66	30	33
Other types	16	50	117	34	60	67
Total	32	100	348	100	90	100

following information for each method: (a) signature, including name, parameters and return type; (b) a string representing the method's body. Using this information provided by the AST, the set C with the defective methods associated to d has been calculated in the following way:

$$C = \{m_i \in M_b \mid \exists\, m_j \in M_a,\ m_i.sig() = m_j.sig(),\ m_i.body() \neq m_j.body()\} \cup \{m_i \in M_b \mid \nexists\, m_j \in M_a,\ m_i.sig() = m_j.sig()\}$$

where $m.sig()$ returns m's signature and $m.body()$ returns a string with m's body. Essentially, C includes the methods m_i in the version before fixing the defect for which there is a method m_j in the version after fixing the defect with the same signature, but with a modified body. Additionally, C includes the methods in the version before the fix that have been removed in the version after fixing the defect.

Finally, we have executed the FindBugs tool in its default configuration over the version of the system before fixing the field defect. The warnings detected in the methods located in the set C have been counted; the remaining warnings have been discarded. This procedure has been based on the following hypothesis: a bug finding tool helps maintainers to detect and fix field defects when it is able to indicate warnings in the methods that must be changed to correct these defects.

2.2 Results

Table 2 presents the number of versions for which FindBugs has raised warnings in the set of methods changed to fix field defects. As shown in this table, in only two out of the 16 versions of the Rhino interpreter FindBugs has been able to issue at least one warning in the set of methods changed to fix the reported field defects. For the `ajc` compiler, FindBugs has issued warnings located in the changed methods for only 11% of the versions evaluated in the experiment. For Lucene, this percentage was 17%.

Figure 1 provides detailed information on the number or warnings raised by FindBugs in the methods changed when fixing the evaluated field defects. In such figures, the x-axis contains the ID (revision number in the version-control system) of the versions with at least one warning in the changed method. The y-axis indicates the number of warnings triggered by FindBugs in such methods. As shown by the figures, in most cases FindBugs has raised just one or two warnings in the set of changed methods (with the exception of two methods from the Lucene system, which have five and eight warnings).

2.2.1 Analysis of the results

By considering the results in Table 2 and Fig. 1, we can conclude that FindBugs has provided minimal indications about the methods responsible for the defects considered in

Table 2 Number of versions with and without warnings in the methods changed to fix field defects

Versions	Rhino		`ajc`		Lucene	
	Qty	%	Qty	%	Qty	%
Changed methods without warnings	14	88	206	89	25	83
Changed methods with at least one warning	2	12	25	11	5	17
Total	16	100	231	100	30	100

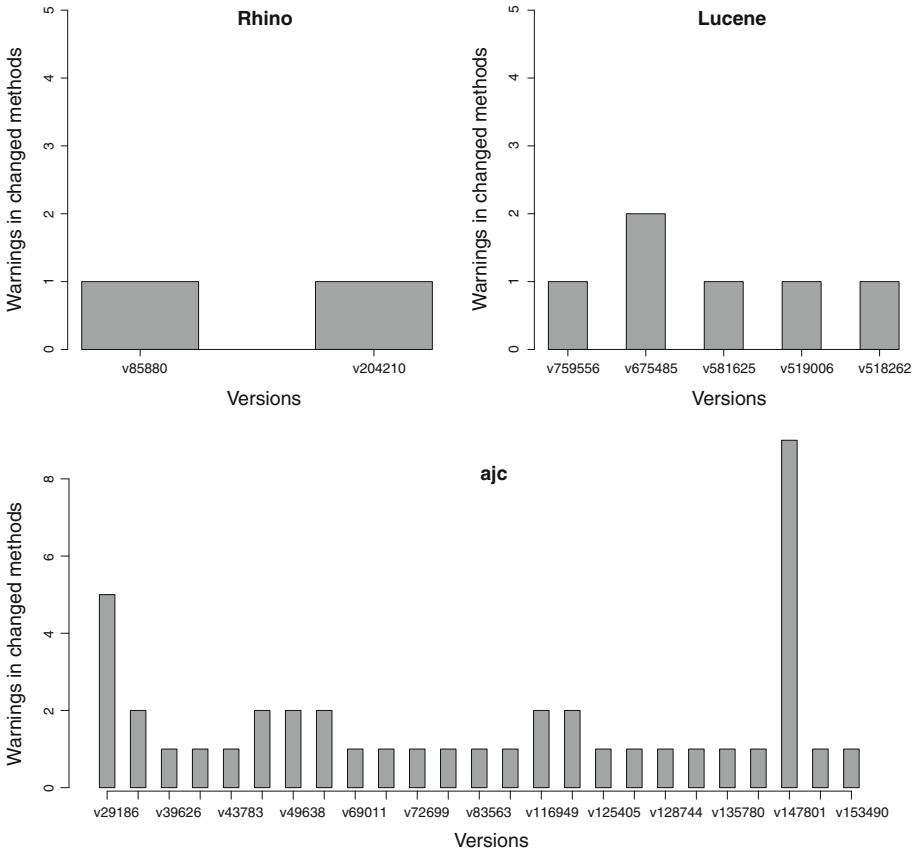


Fig. 1 Number of warnings in changed methods

the experiment. Based on such results, our answer to question Q1 is negative: we have not observed evidences toward a static correspondence between field defects and warnings reported by FindBugs. More specifically, even relying on a weak precondition for matching warnings to field defects (i.e., a definition that does not check whether the warnings semantically relate to the changes applied to fix a defect), FindBugs has not been able to raise warnings in most of the defective methods. In other words, the warnings triggered by the tool would not have helped the maintainers to detect, understand, and remove the field defects evaluated in the study.

2.2.2 Precision and recall

To complement our previous analysis, we also measured the precision and recall of the warnings raised by FindBugs. By measuring precision, our goal was to provide information on the number of false positives raised by FindBugs, i.e., methods with warnings but that have not been changed to fix bugs. On the other hand, by measuring the recall our intention was to show information on the number of false negatives, i.e., the absence of warnings in methods changed to fix defects. First, we measured precision at the method level in the following way:

Table 3 Precision and recall

	# Warnings	# Warnings/ KLOC	Precision (%)	Recall (%)
Rhino (16 versions)				
Max	119	3.9	33.3	50.0
Min	101	2.6	0.0	0.0
Mean	112.6	3.6	3.3	3.8
Median	113.0	3.7	0.0	0.0
SD	5.5	0.3	9.4	12.6
`ajc` (206 versions)				
Max	938	11.7	100.0	100.0
Min	225	7.0	0.0	0.0
Mean	631.0	9.8	5.6	6.9
Median	813.0	9.9	0.0	0.0
SD	267.9	0.7	19.6	22.4
Lucene (30 versions)				
Max	205	6.9	100.0	100.0
Min	118	4.4	0.0	0.0
Mean	152.7	6.3	6.8	3.7
Median	152	6.3	0.0	0.0
SD	30.8	0.5	20.5	18.2

$$\text{precision} = \frac{\text{number of changed methods with at least one warning}}{\text{number of methods with at least one warning}}$$

To measure recall, we have considered a method as relevant when it has been changed to fix a field defect. Moreover, we consider that FindBugs detects a relevant method when it raises at least one warning in such method. Based on these assumptions, we have calculated recall in the following way:

$$\text{recall} = \frac{\text{number of changed methods with at least one warning}}{\text{number of changed methods}}$$

Table 3 shows the values measured for recall and precision for the systems considered in this first experiment. Considering the versions evaluated in the experiment, this table reports the following results: maximum value (Max), minimum value (Min), mean, median, and standard deviation (Std Dev). Three conclusions can be derived from such results:

- FindBugs raises a large number of warnings. We have counted 3.6 ± 0.3, 9.8 ± 0.7, and 6.3 ± 0.5 warnings/KLOC for the evaluated versions of the systems Rhino, `ajc`, and Lucene, respectively ($a \pm s$ means average a with standard deviation s).
- Due to the large number of warnings, the precison results are extremely low: $3.3 \pm 9.4\%$, $5.6 \pm 19.6\%$, and $6.8 \pm 20.5\%$, for the systems Rhino, `ajc`, and Lucene, respectively.
- As another consequence of the large number of warnings, the recall results have also been low: $3.8 \pm 12.6\%$, $6.9 \pm 22.4\%$, $3.7 \pm 18.2\%$, for the systems Rhino, `ajc`, and Lucene, respectively.

3 Correlation

To correlate warnings to defects, we have considered 30 systems from the Apache Foundation. Table 4 presents detailed information about our target systems. The numbers about lines of code have been obtained using the JavaNCSS tool.[4] The presented results consider only classes in the core packages of each system (i.e., packages matching `org.apache.[system].*`) and ignore comments and blank lines. In total, the systems evaluated in this second experiment have almost one and a half million lines of code. Furthermore, they meet the following criteria: (a) they represent medium-sized to large and complex systems (the smallest system has 9.8 KLOC and the largest one has 178.8 KLOC); (b) their bytecode (in the format of a JAR file) is publicly available; (c) they have a well-documented history of defects.

The following tasks have been performed to collect the data necessary for the correlation study:

1. We have downloaded the JAR file of each of the systems considered in the study (this file has been downloaded from the software repository of the Apache Foundation).
2. We have executed FindBugs two times on each of the JAR files downloaded in the previous step. In the first execution, we have relied on FindBugs default configuration. In the second execution, the tool has been configured to just report high-priority warnings, i.e., warnings denoting code with a high probability to lead to defects. The total number of warnings reported on each of such executions has been collected.
3. We have accessed the Jira bug tracking system used by the Apache Foundation to collect the number of defects for the systems considered in the study. Basically, we used Jira to collect the number of defects (or bugs) that have been fixed in the first version after the version considered in the study. For example, for Lucene, we have counted the number of warnings on version 2.3.2. Therefore, we used the Jira system to get the number of defects fixed in version 2.4 (the next version, after 2.3.2). In this way, we can assure that the defects considered in the study actually existed in the versions used to count the number of warnings.

Table 5 presents the following data collected after the aforementioned tasks: number of warnings reported by FindBugs in its default configuration (column FB), number of high-priority warnings reported by FindBugs (column FBH), number of defects fixed in the first version after the version considered in the study (column BUGS). Finally, the last three columns of Table 5 show the density of the reported warnings, i.e., the values at columns FB, FBH, and BUGS divided by the number of thousands of lines of source code (KLOC) of the respective versions. Instead of absolute values, the goal is to correlate data that take into account the size of the systems evaluated in the study.

Table 6 presents general statistics about the collected data. As can be observed in this table, on average, FindBugs has reported 6.11 warnings and 1.57 high-priority warnings per KLOC, i.e., the number of general warnings is almost four times greater than the number of high-priority warnings.

3.1 Spearman's rank correlation test

Spearman's rank correlation test is a measure for the statistical correlation between two variables (Sprent and Smeeton 2007). The test yields a coefficient between −1 (perfect

[4] http://www.kclee.de/clemens/java/javancss.

Table 4 Systems used to correlate warnings and defects

	System	Version	Date	LOC	Description
1	Struts2	2.0.6	18/02/07	14,466	Web application framework
2	CXF	2.2	18/03/09	106,225	Web service framework
3	ApacheDS	1.5.0	05/04/07	41,916	LDAP-based directory service
4	Jackrabbit	1.4	15/01/08	92,519	Content repository manager
5	Myfaces Core	1.2.0	17/07/07	20,185	JSF implementation
6	Myfaces Tomahawk	1.1.7	13/09/08	58,462	Custom JSF components
7	OpenJPA	1.0.0	23/08/07	101,787	Persistence API
8	Tuscany SCA	1	19/09/07	41,081	SOA-based framework
9	UIMA	2.1.0	07/03/07	80,763	Unstructured information mnger
10	Wicket	1.4.0	16/07/09	50,714	Web application framework
11	Hadoop Common	0.16.4	05/05/08	61,210	Utilities for Hadoop projects
12	Hadoop Hbase	0.2.0	05/08/08	30,504	Distributed database system
13	Ivy	2.1.0	08/10/09	25,779	Dependency manager system
14	James Server	2.2.0	16/06/04	15,499	Mail enterprise server
15	Lucene	2.3.2	06/05/08	44,585	Information retrieval library
16	Roller	3.1	23/04/07	30,879	Blog server
17	Shidig	1.1b1	22/07/09	18,679	OpenSocial container
18	Solr	1.3	12/09/08	28,656	Text search platform
19	Tapestry	4.1.2	27/06/07	31,772	Web application framework
20	Axis2	1	05/05/06	27,102	Web service framework
21	Geronimo	2.1	18/02/08	38,358	Application server
22	Xalan	2.6.0	29/02/04	46,213	XSLT implementation
23	Xerces	2.9.1	14/09/07	68,953	XML parser
24	Beehive	1	30/09/05	49,130	Java application framework
25	Derby	10.1.2.1	18/11/05	178,880	Relational database system
26	Cayenne	2.0.2	18/01/07	62,800	Object/Relational framework
27	XMLBeans	2.0.0	28/06/05	71,515	XML Parser
28	JDO	2.0.0	04/04/06	15,943	Persistence API
29	DdlUtils	1.0.0	28/06/07	10,541	Database Definition (DDL) API
30	iBatis	2.3.0	28/11/06	9,839	Object/Relational framework

negative correlation) and +1 (perfect positive correlation). Moreover, the test does not measure the correlation between the absolute values of the involved variables, but the correlation on the order of those values in a rank. The main advantage of Spearman is that it can be applied to any data sample, i.e., it does not require the sample to meet a particular distribution (Pfleeger 1995).[5]

Table 7 presents the Spearman's coefficients expressing the correlation between the two variables considered in this work: field defects (represented by the variable BUGS/KLOC)

[5] Indeed, we have applied the Shapiro-Wilk test to check whether the number of warnings and the number of defects reported in Table 5 follow a normal distribution. Because the test rejected the assumption of normality at the level of 5% of significance, we decided to use the Spearman test to measure correlation.

Table 5 Absolute and relative data about warnings and defects

	System	FB	FBH	BUGS	FB/KLOC	FBH/KLOC	BUGS/KLOC
1	Struts2	62	15	35	4.29	1.04	2.42
2	CXF	644	133	40	6.06	1.25	0.38
3	ApacheDS	332	114	61	7.92	2.72	1.46
4	Jackrabbit	400	65	109	4.32	0.70	1.18
5	Myfaces Core	81	10	43	4.01	0.50	2.13
6	Myfaces Tomahawk	182	43	25	3.11	0.74	0.43
7	OpenJPA	494	75	45	4.85	0.74	0.44
8	Tuscany SCA	242	63	22	5.89	1.53	0.54
9	UIMA	466	211	77	5.77	2.61	0.95
10	Wicket	163	29	12	3.21	0.57	0.24
11	Hadoop Common	421	224	138	6.88	3.66	2.25
12	Hadoop Hbase	343	95	35	11.24	3.11	1.15
13	Ivy	145	16	47	5.62	0.62	1.82
14	James Server	106	30	104	6.84	1.94	6.71
15	Lucene	461	121	66	10.34	2.71	1.48
16	Roller	378	57	87	12.24	1.85	2.82
17	Shidig	33	6	7	1.77	0.32	0.37
18	Solr	208	104	226	7.26	3.63	7.89
19	Tapestry	129	16	27	4.06	0.50	0.85
20	Axis2	183	44	82	6.75	1.62	3.03
21	Geronimo	296	72	53	7.72	1.88	1.38
22	Xalan	573	233	217	12.40	5.04	4.70
23	Xerces	196	60	49	2.84	0.87	0.71
24	Beehive	375	51	91	7.63	1.04	1.85
25	Derby	978	125	104	5.47	0.70	0.58
26	Cayenne	293	38	31	4.67	0.61	0.49
27	XMLBeans	290	173	9	4.01	2.42	0.13
28	JDO	98	25	78	6.15	1.57	4.89
29	DdlUtils	46	3	18	4.36	0.28	1.71
30	iBatis	55	3	18	5.59	0.30	1.83

Table 6 Characterization of the data collected about warnings and field defects

	FB	FBH	BUGS	KLOC	FB/KLOC	FBH/KLOC	BUGS/KLOC
Max	978	233	226	178.88	12.40	5.04	7.89
Min	33	3	7	9.84	1.77	0.28	0.13
Mean	289.10	75.13	65.20	49.17	6.11	1.57	1.89
Median	266	58.50	48.00	41.50	5.70	1.15	1.42
SD	210.14	66.25	54.11	36.20	2.67	1.20	1.89

and warnings reported by FindBugs (represented by the following variables: FB/KLOC and FBH/KLOC). The coefficients reported in this table have a significance level of at least 95% (p-value ≤ 0.05) and they have been calculated using the R statistical tool.

Table 7 Spearman coefficients

	FB/KLOC	FBH/KLOC
BUGS/KLOC	0.57	0.36
p-value	0.00	0.05

3.1.1 Analysis of the results

As presented in Table 7, the results of the Spearman's test show a statistical correlation between the two variables considered in the study (field defects and warnings reported by FindBugs). Essentially, when considering only high-priority warnings, the Spearman's coefficient was 0.36. When the correlation is extended to consider all kinds of warnings, this coefficient has increased to 0.57. In other words, such degrees of correlation suggest that warnings—specially default warnings—can be viewed as early indicators of the quality of a system.

4 Lessons learned

This section discusses the main lessons learned on the study. First, although the experiment about static correspondence has involved only three systems, it provides evidences that FindBugs does not help to detect defective program elements. The main reason is that there is a wide spectrum of field defects that can be reported for a system. Additionally, many defects are related to logic or semantic errors (i.e., errors due to wrong results or unexpected behavior). For example, in the specific case of Rhino and `ajc` most errors are due to source code in JavaScript or AspectJ that is not processed as expected. On the other hand, the warnings reported by FindBugs are quite specific. Most of them are centered on violations of recommending programming patterns (e.g., classes that implement `hashCode` must also implement an `equals` method) and errors detected by local data and control flow analysis (e.g. null pointer access).

Unlike the first experiment, the second study on correlation has presented positive results. First, the literature describes studies assessing the application of static analysis tools in real life systems. However, usually they focus on a single system (e.g., the Windows operating system (Nagappan and Ball 2005) or SAP/R3 (Holschuh et al. 2009)). Unlike these studies, in this article, we have evaluated thirty medium-sized systems, maintained by the same organization (the Apache Foundation). Second, the experiment about correlation has produced valuable data on the density of warnings reported by FindBugs and on the density of field defects. For example, we have not observed extreme densities of high-priority warnings (on average, 1.57 warnings per KLOC). Therefore, this result rebuts the common criticism about the massive number of warnings generated by static analysis tools (Kim and Ernst 2007; Wagner et al. 2005). More specifically, we learned that the number of warnings can be managed when the tool is properly configured to raise only high-priority warnings.

Finally, as another result of the study, we have observed a moderate level of correlation between the number of warnings reported by FindBugs and the number of defects reported by the users of the systems evaluated in the article. Particularly, the results obtained through the Spearman's rank correlation test suggest that systems with more warnings are subjected to present more field defects after being released for use. Therefore, we conclude

that bug finding tools like FindBugs can play an important role on assessing the quality of the versions of a system.

5 Threats to validity

In this section, we discuss potential threats to the validity of our study. As usual, we have arranged possible threats in three categories: external, internal, and construct validity (Perry et al. 1997):

5.1 External validity

This form of validity refers to the degree to which we can extend the results of a study to a wider population. In the study about static correspondence, we have considered three systems. Particularly, Rhino and `ajc` were the only systems available at the iBugs repository when the study has been realized. The reason for having only two systems in this repository is simple: recovering the source code before and after fixing a given field defect is not a trivial task, since in most systems we do not have a direct traceability between bugs and release identifiers (Dallmeier and Zimmermann 2007). However, even by evaluating just three systems, it was possible to assert the existence of an important gap between the warnings generated by FindBugs and the large spectrum of field defects. This gap explains the limited degree of static correspondence observed at Sect. 2.

Regarding the correlation study, it has been based on thirty systems from the Apache Foundation, totaling almost one and a half million lines of source code. This sample is one of the strengths of our study because besides including a credible number of systems, the selected systems represent real-world and non-trivial applications, with a consolidated number of users and a relevant history of bugs. On the other hand, our sample may be not representative of industrial, non-open source-based systems.

Finally, both experiments have been based on a single bug finding tool. Moreover, they should not be generalized to systems implemented in other languages, including unsafe static languages (such as C and C++) or dynamic languages (such as Ruby or Python).

5.2 Internal validity

This form of validity evaluates whether the study findings are due to factors that have not been controlled or measured in the experiment. In the case of the static correspondence study, this risk is minimized because iBugs has been carefully constructed to provide benchmarks for bug finding tools. However, we have not validated our classification of the requests as corrective maintenance with the developers of the evaluated systems. Instead, we have relied on the description provided by the end-users when opening the requests in the Bugzilla tracking system. Although we acknowledge the absence of this validation as a possible threat to the internal validity of the study, we have confidence on the proposed classification, because end-users—at least in our specific study—usually provide very detailed descriptions about their requests (presumably to maximize the chances of the requests been processed quickly).

In the case of the correlation study, it was not possible to control the number of users that accessed the versions considered in the study. This variable is important because in systems with a large base of users, the number of field defects tends to be higher than in systems with a reduced number. In other words, the ideal scenario would include only

systems accessed by the same number of users, with a similar profile (experience users, novice users and so on), with each user accessing the systems the same number of hours per day. Clearly, setting this ideal scenario for real systems is not trivial. Therefore, to control this threat, we have restricted our sample to well-know systems from a single software organization. Such systems have a relevant base of users, most of them having a similar profile (in general, the users of Apache systems are software developers).

5.3 Construct validity

This form of validity assesses whether the study findings are not due to misleading data or errors committed during the experiment. In the study about static correspondence, we have carefully filtered out field defects, in order to discard requests associated, for example, with adaptive or perfective maintenance. A similar procedure has been followed in the second part of the paper. However, in this case, it was easier to filter the defects, because we have only considered defects explicitly classified as "fixed bugs" in the documentation available at Jira.

6 Related work

Related work can be arranged in three groups: (a) studies on the relevance of bug finding tools; (b) correlation studies including bug finding tools and (c) correlation studies on software metrics.

6.1 Studies on the relevance of bug finding tools

Wagner et al. have evaluated the effectiveness of bug finding tools in two large systems (Wagner et al. 2008). In their work, they have considered two tools: FindBugs and PMD. Similar to our work, their goal was to assess the effectiveness of such tools to detect defects that occur in the field. For the first evaluated system, they could not find a single warning related to 72 reported field defects. For the second system, they found a causal relation between four warnings and field defects. Therefore, their results are compatible with the ones we have obtained in our first experiment. In fact, we have showed that neither a weaker relation—as our notion of static correspondence—can be established between warnings and the program elements changed to fix field defects.

Zheng et al. have followed the GQM paradigm to determine "whether automated static analysis can help an organization to economically improve the quality of software products" (Zheng et al. 2006). Particularly, they have showed that the number of warnings raised by static analysis tools can be a fairly good indicator of fault prone modules, which is essentially the same conclusion we have reached on our second study, described in Sect. 3.

In a recent work, we have evaluated the lifetime of the warnings reported by FindBugs when executed over five stable releases of the Eclipse platform (Araujo et al. 2011). We have concluded that the number of false positives (by our criteria, warnings not removed in subsequent releases) can be reduced to less than 50% when FindBugs is configured to just trigger high-priority warnings. However, for the purpose of locating field defects, restricting the analysis to high-priority warnings will just undermine the results of the first experiment described in this article. Essentially, because we have not observed high levels of static correspondence relying on FindBugs default configuration, we can infer that such levels would be even lower if we had used a restricted set of warnings.

6.2 Correlation studies including bug finding tools

Similar to our study on correlation (Sect. 3), Nagappan and Ball have described an experiment to measure correlation between warnings reported by static analysis tools and defects (Nagappan and Ball 2005). By also using Spearman's test, they have found a positive correlation between the density of warnings issued by the PREfix/PREfast tools and the density of pre-release defects detected in the Windows Server 2003. However, there are three main differences between their work and the study reported in the current article: (a) they do not consider field defects, but pre-release defects (i.e., defects found by developers before the release of a system, using, for example, tests or manual inspections); (b) they have analyzed a single system (Windows Server 2003), while in our study we have assessed thirty systems; (c) they relied on PREfix/PREfast tools for finding potential defects in systems implemented in C (a non-type safe language), while we have centered our study on Java-based systems. In spite of such differences, we consider that our work complements Nagappan and Ball's experiment by showing that warnings are correlated not only to pre-release defects (as suggested by their experiment), but also to post-release defects (as suggested by our second experiment).

Butler et al. concluded for the existence of correlation between warnings raised by FindBugs and violations in the name patterns of identifiers, using a sample of eight systems (although this correlation has ceased when they restricted the analysis to high-priority warnings) (Butler et al. 2009). Our claim is that such correlation is due to poor quality code, as the correlation we have measured in our second experiment. Stated otherwise, our claim is that there is a broad event—namely poorly designed and implemented code—that is the common cause behind the following correlations: (a) between name patterns violations and FindBugs' warnings (as discussed in Butler's works); (b) between field defects and FindBugs' warnings (as discussed in the second experiment reported in this paper).

6.3 Correlation studies including number of defects

There are also studies that have investigated the relation between the number of defects and the following properties of OO systems:

- **Metrics**: Subramanyam and Krishnan have investigated the relation between defects and classical metrics, such as CBO, WMC, and DIT (Subramanyam and Krishnan 2003). In their study, they have evaluated a single e-commerce system, with modules implemented in C++ and Java. For the part of the system implemented in C++, they concluded that WMC, DIT, and CBO*DIT have had a relevant impact on the number of defects. For the modules implemented in Java, only CBO*DIT has had an impact on defects. Similarly, Nagappan et al. have conducted a study on five components of the Windows operating system in order to investigate the relationship between complexity metrics and field defects (Nagappan et al. 2006). Later, the study has been replicated to consider a large ERP system (SAP R3) (Holschuh et al. 2009). They have also measured a significant correlation between complexity metrics and field defects.
- **Design Flaws:** D'Ambros et al. have demonstrated a relationship between well-known software design flaws (e.g. Brain Method, Feature Envy, and Shotgun Surgery (Lanza and Marinescu 2006)) and post-release defects (D'Ambros et al. 2010). In another work, the authors showed the existence of an important correlation between field defects and two metrics they have proposed, called churn of source code and entropy of source code (D'Ambros et al. 2010).

To conclude, the mentioned studies suggest that the correlation investigated in our work can be extended to consider other variables, mainly metrics designed to evaluate the static structure and the overall design quality of object-oriented systems.

7 Conclusions

In this article, we have investigated the role that bug finding tools can play in software quality assurance. Our investigation has been directed to answer two questions: (Q1) Is there a correspondence between the static location of warnings and the program elements changed in order to remove field defects? (Q2) Is there a statistical relation between warnings and field defects? Our answer to Q1 was negative: after matching warnings to defective program elements in three medium-to-large size systems, we have not observed evidences toward a static correspondence between post-release defects and warnings reported by FindBugs. On the other hand, our answer to Q2 was positive: by analyzing thirty systems from the Apache Foundation, we have detected a moderate correlation between field defects and warnings reported by FindBugs. Therefore, FindBugs has not provided static correspondence with actual bug fixes at the method level of granularity. However, when we lifted the analysis to the level of projects, we have observed that those projects with a higher density of warnings have also presented a higher density of defects.

At a first view, these answers seem to contradict each other. However, we believe that this apparent contradiction can be explained as follows. First, our first experiment provides strong indications that warnings do not directly contribute to locate the software components responsible for field defects. However, warnings seem to be good indicators for the internal quality of a software system, mainly in terms of adherence to recommended programming practices and correct use of standard libraries (as indicated by previous studies (Wagner et al. 2008)). In other words, our second experiment suggests that poor quality code tends to present both more warnings and defects, although there is not a direct causal connection between these two variables. However, we acknowledge that the reported experiments should be extended and replicated to provide more robust conclusions on this subject. It is also important to investigate the reasons for the mismatch observed between the static location of warnings and defects.

In the future, we also intend to investigate other statistical techniques to correlate warnings to defects, such as Granger's causality test (Granger 1969).

Acknowledgments This work was supported by FAPEMIG, CAPES, and CNPq.

References

Araujo, J. E., Souza, S., & Valente, M. T. (2011). Study on the relevance of the warnings reported by Java bug finding tools. *IET Software, 5*(4), 366–374.

Ayewah, N., Hovemeyer, D., Morgenthaler, J. D., Penix J., & William, P. (2008). Using static analysis to find bugs. *IEEE Software, 25*(5).

Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D. (2010). A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM, 53*(2), 66–75.

Butler, S., Wermelinger, M., Yu, Y., & Sharp, H. (2009). Relating identifier naming flaws and code quality: An empirical study. In *16th working conference on reverse engineering (WCRE)*, pp. 31–35.

Copeland, T. (2005). *PMD applied*. Alexandria: Centennial Books.

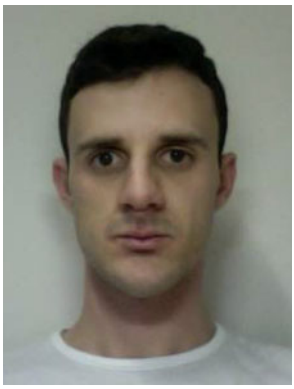
Dallmeier, V., & Zimmermann, T. (2007). Extraction of bug localization benchmarks from history. In *22th conference on automated software engineering (ASE)*, pp. 433–436.

D'Ambros, M., Bacchelli, A., & Michele, L. (2010). On the impact of design flaws on software defects. In *10th international conference on quality software (QSIC)*, pp 23–31.
D'Ambros, M., Lanza, M., & Robbes, R. (2010). An extensive comparison of bug prediction approaches. In *7th working conference on mining software repositories (MSR)*, pp. 31–41.
Foster, J. S., Hicks, M. W., & Pugh, W. (2007). Improving software quality with static analysis. In *7th workshop on program analysis for software tools and engineering (PASTE)*, pp. 83–84.
Granger, C. W. J. (1969). Investigating causal relations by econometric models and cross-spectral methods. *Econometrica, 37*(3), 424–438.
Holschuh, T., Pauser, M., Herzig, K., Zimmermann, T. P., & Rahul, Z. (2009). Andreas predicting defects in sap java code: An experience report. In *31st international conference on software engineering (ICSE)*, pp. 172–181.
Hovemeyer, D., & Pugh, W. (2004). Finding bugs is easy. *SIGPLAN Notices*, *39*(12), 92–106.
Johnson, S. C. (1977). Lint: A C program checker. Technical Report 65, Bell Laboratories.
Kim, S., & Ernst, M. D. (2007). Which warnings should I fix first? In *15th international symposium on foundations of software engineering (FSE)*, pp. 45–54.
Lanza, M., & Marinescu, R. (2006). Object-oriented metrics in practice. Springer.
Larus, J. R., Ball, T., Das, M., DeLine, R., Fahndrich, M., Pincus, J., Rajamani, S. K., Ramanathan, V. (2004). Righting software. *IEEE Software*, *21*(3), 92–100.
Louridas, P. (2006). Static code analysis. *IEEE Software*, *23*(4), 58–61.
Nagappan, N., & Ball, T. (2005). Static analysis tools as early indicators of pre-release defect density. In *27th international conference on software engineering (ICSE)*, pp. 580–586.
Nagappan, N., Ball, T., & Zeller, A. (2006). Mining metrics to predict component failures. In *28th international conference on software engineering (ICSE)*, pp. 452–461.
Perry, D. E., Porter, A. A., & Votta, L. G. (1997). A primer on empirical studies (tutorial). In *Tutorial presented at 19th international conference on software engineering (ICSE)*, pp. 657–658.
Pfleeger, S. L. (1995). Experimental design and analysis in software engineering, part 5: Analyzing the data. *Software Engineering Notes*, *20*(5), 14–17.
Sprent, P., & Smeeton, N. C. (2007). *Applied nonparametric statistical methods*. Boca Raton: Chapman & Hall.
Subramanyam, R., & Krishnan, M. S. (2003). Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects. *IEEE Transaction on Software Engineering*, *29*(4): 297–310.
Wagner, S., Jürjens, J., Koller, C., & Trischberger, P. (2005). Comparing bug finding tools with reviews and tests. In *17th international conference on testing of communicating systems (TestCom)*, volume 3502 of *LNCS*, pp. 40–55. Springer.
Wagner, S., Aichner, M., Wimmer, J., & Schwalb, M. (2008). An evaluation of two bug pattern tools for Java. In *1st international conference on software testing, verification, and validation (ICST)*, pp. 248–257.
Zheng, J., Williams, L., Nagappan, N., Hudepohl, J. P., & Vouk M. A. (2006). On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, *32*(4).

Author Biographies

Cesar Couto is a PhD student in the Computer Science Department at the Federal University of Minas Gerais, Brazil. He is also a lecturer in the Department of Computing at CEFET-MG, Brazil. His research interests include software maintenance and evolution, software quality, and programming languages. Couto has an MSc in Computer Science from the Federal University of Minas Gerais.

Joao Eduardo Montandon is a MSc student in the Computer Science Department at the Federal University of Minas Gerais, Brazil. His research interests include software quality, software comprehension, and software engineering issues for mobile application development.



Christofer Silva is an undergraduate student in Computer Engineering at the Department of Computing at CEFET-MG, Brazil. His research interests include software maintenance and evolution, software quality, and programming languages.

Marco Tulio Valente is an assistant professor in the Computer Science Department at the Federal University of Minas Gerais, Brazil. His research interests include software maintenance and evolution, software quality, and software modularization and remodularization. Valente has a PhD in Computer Science from the Federal University of Minas Gerais. He is a member of the ACM, the IEEE Computer Society, and the Brazilian Computer Society.