Prioritizing Warning Categories by Analyzing Software History

Sunghun Kim, Michael D. Ernst
Computer Science & Artificial Intelligence Lab (CSAIL)
Massachusetts Institute of Technology
{hunkim, mernst}@csail.mit.edu

Abstract

Automatic bug finding tools tend to have high false positive rates: most warnings do not indicate real bugs. Usually bug finding tools prioritize each warning category. For example, the priority of "overflow" is 1 and the priority of "jumbled incremental" is 3, but the tools' prioritization is not very effective.

In this paper, we prioritize warning categories by analyzing the software change history. The underlying intuition is that if warnings from a category are resolved quickly by developers, the warnings in the category are important. Experiments with three bug finding tools (FindBugs, JLint, and PMD) and two open source projects (Columba and jEdit) indicate that different warning categories have very different lifetimes. Based on that observation, we propose a preliminary algorithm for warning category prioritizing.

1. Introduction

Bug finding tools such as FindBugs [6], JLint [1], and PMD [4] analyze source or binary code and warn about potential bugs. These tools tend to have a high rate of false positives: most warning instances do not indicate real bugs. These tools usually prioritize warning categories to put likely false positives at the bottom of the list, but these tools' prioritization is not very effective [9].

We use the software change history to prioritize warning categories. (Often warning instances from bug finding tools have categories such as *overflow* or *NP always Null.*) Suppose a software change would cause a bug finding tool to issue a warning instance from the o*verflow* category. If a developer found the underlying problem and fixed it quickly, the warning category is probably important. (We do not assume the software developer is necessarily using the bug finding tool.) On the other hand, if a software change introduced a warning instance that was not removed for a long time, then warnings of that category may be neglectable, since the problem was not noticed or was not considered worth fixing.

Using a version control system that stores a project's change history, it is possible to find out when a file was changed or fixed [8]. We noticed that different warning categories have different lifetimes over software history. We list the most short-lived warning categories (which may be most important or useful) and the most long-lived warning categories. The warning category priorities assigned by bug finding tools do not match the priorities assigned based on warning category lifetimes. Prioritizing warning categories based on their lifetimes may help to make bug detection tools more useful.

Our work aggregates properties of warning instances to prioritize warning categories. It does not give different priorities to two different instances from the same category. For example, warning instances from the *overflow* category have the same priority. This suggests that our technique will be most effective when the categories are relatively fine-grained and homogeneous (with respect to their importance and lifetime).

2. Background and Related Work

We introduce three bug finding tools briefly and discuss related work in this section.

PMD finds syntactic error patterns from source code [4]. JLint analyzes Java bytecode and performs syntactic checking and data flow analysis to find potential bugs [1]. FindBugs also analyzes Java bytecode to find pre-defined error [6].

Ruter et al. [10] compared bug finding tools for Java including PMD, JLint, and FindBugs. They analyze overlapping warning categories and category correlation. However, our approach prioritizes warning categories using change history while their work compares tools and finds similarities and differences among tools.

Spacco et al. [11] observed FindBugs warning categories across software versions. They measure lifetimes of warning categories, warning number rates, and the degree of decay using warning numbers. This research is similar to ours in that they observe warning category trends over software history. However, they observe warning categories in very coarse grained software versions (in releases) while we observe categories in each version control system transaction.

Fourth International Workshop on Mining Software Repositories (MSR'07)
0-7695-2950-X/07 $20.00 © 2007 IEEE

Kremenek and Engler [9] prioritize warning categories using frequency of defects. Similarly, Boogerd and Moonen [3] use execution likelihood analysis to prioritize warning instances. Our approach prioritizes warnings analyzing software history.

3. Experiment Setup

Table 1 gives information about our subject programs, JEdit [1] and Columba [2]. The Kenyon infrastructure is used to manage software history [2]. Kenyon checks out the software transaction in the software change history. Then it compiles each transaction and creates a jar file or class files. Kenyon ignores transactions which cannot be compiled. For example, Kenyon only use 1,486 compilable transactions out of a total of 1,703 transactions for Columba. Using three bug finding tools (FindBugs, JLint, and PMD), Kenyon gets warning instances from each software transaction.

Table 1. Analyzed Projects.

Project	Software type	Period	# of compilable transaction	# of transaction
Columba	Email Client	11/2002 ~ 06/2004	1,486	1,703
jEdit	Editor	09/2001 ~ 11/2006	1,200	1,509

4. Experiment and Reprioritized Warning Categories

We measure the lifetime of each warning category over software change history. Based on the lifetime, we prioritize each warning category.

Whenever a warning instance appears in the software change history, that time and its category are marked. When the warning instance disappears, the time difference between the marked time and the current transaction time is the lifetime of the warning category. If a warning instance is never fixed, we give a penalty to the lifetime by adding 365 days. We then compute the average lifetime of instances in each category.

The warning categories and their lifetimes of the two projects are shown in Figure 1 and Figure 2. Each bar in Figure 1 and Figure 2 presents a warning category. In Columba, some warning categories have a relatively short lifetime as marked in Figure 1. We assume warning instances in these categories are more serious than others. Similarly, some warning categories in Columba have relatively long lifetimes, and warnings in these categories are less likely to be real bugs.

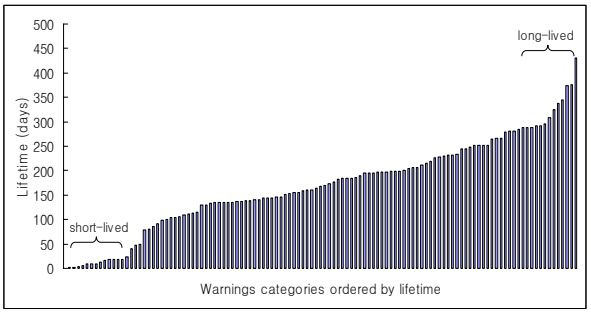

Figure 1. Warning category lifetimes of Columba.

JEdit warning category lifetimes in Figure 2 have similar properties: some have very short lifetimes while some have long lifetimes.

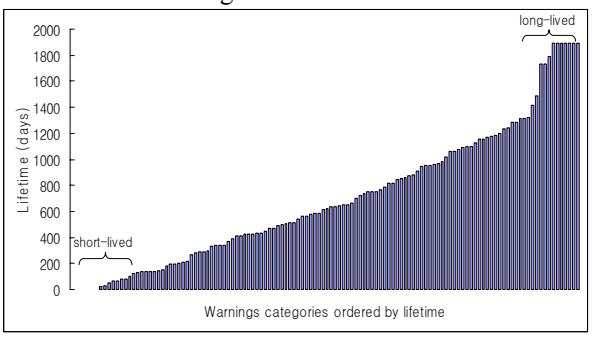

Figure 2. Warning category lifetimes of jEdit.

After computing lifetimes for all warning categories, we order categories by their lifetime. The top 10 short-lived and long-lived warning categories with detailed information are listed in Table 2 and Table 3. The ranks are given based on the lifetimes. The priorities shown in Table 2 and Table 3 are given by the corresponding bug finding tool. For example, in Columba, the 7th ranked warning category is *SA FIELD SELF ASSIGNMENT* (the priority given by FindBugs is 1) occurs 8 times and is fixed within 9 days on average. The high priority such as 1 given by bug finding tools does not necessarily mean that the warning category is a short-lived one. For example, the priority of *Weak cmp* (rank 108) of JLint is 1, but its lifetime is very long, 291 days. Note that FindBugs sometimes gives different priorities to the same category (see the rank 6 and 7 in Columba).

Are warning category lifetimes between two projects correlated? We measured the Pearson's correlation coefficient between the common warning category lifetimes shown in the two projects. The coefficient, r, is 0.218 which indicates the category lifetimes between the two projects have no correlation.

[1] The jEdit project: http://www.jedit.org/
[2] The Columba project: http://www.columbamail.org/

Table 2. Columba top 10 short-lived and long-lived warning categories ordered by their lifetime.

Rank	Tool	Tool Priority (1 is high)	Category Group	Category	Occurr-ence	Lifetime (days)	
						Average	Standard deviation
1	FindBugs	1	CORRECTNESS	NP NULL PARAM DEREF NONVIRTUAL	1	0.16	0
2	JLint	1	wait_nosync	Wait nosync	1	1.0	0
3	PMD	3	Basic Rules	Unconditional If Statement	1	1.1	0
4	FindBugs	1	BAD PRACTICE	DE MIGHT IGNORE	1	2.8	0
5	FindBugs	2	STYLE	SF SWITCH FALLTHROUGH	2	6.2	0.16
6	FindBugs	2	CORRECTNESS	SA FIELD SELF ASSIGNMENT	2	8.9	0
7	FindBugs	1	CORRECTNESS	SA FIELD SELF ASSIGNMENT	8	8.9	0
8	FindBugs	2	STYLE	SA LOCAL SELF ASSIGNMENT	2	8.9	0
9	FindBugs	2	STYLE	IM BAD CHECK FOR ODD	2	13.4	9.44
10	JLint	1	zero_result	Zero result	2	17.1	0
108	JLint	1	weak_cmp	Weak cmp	98	291.1	27.29
109	FindBugs	1	MALICIOUS CODE	MS MUTABLE ARRAY	3	292.1	23.12
110	FindBugs	1	CORRECTNESS	NP NONNULL RETURN WARNING	2	295.9	190.21
111	FindBugs	2	STYLE	BC UNCONFIRMED CAST	94	308.5	29.66
112	JLint	1	Not overridden	Hashcode not overridden	12	324.1	19.43
113	PMF	3	Basic Rules	Override Both Equals And Hashcode	10	337.4	2.07
114	FindBugs	2	CORRECTNESS	NP GUARANTEED DEREF	3	345.1	126.86
115	FindBugs	2	BAD_PRACTICE	OS OPEN STREAM	9	374.7	10.57
116	FindBugs	2	BAD_PRACTICE	SE BAD FIELD STORE	3	376.7	81.81
118	FindBugs	1	CORRECTNESS	EC UNRELATED TYPES	2	431.0	113.73

Table 3. jEdit top 10 short-lived and long-lived warning categories ordered by their lifetime.

Rank	Tool	Tool Priority (1 is high)	Category Group	Category	Occurr-ence	Lifetime (days)	
						Average	Standard deviation
1	FindBugs	1	CORRECTNESS	NP ALWAYS NULL	1	0.01	0
2	FindBugs	1	CORRECTNESS	NP NULL PARAM DEREF NONVIRTUAL	1	0.02	0
3	JLint	1	Bounds	Bad index	1	0.028	0
4	FindBugs	1	BAD_PRACTICE	DE MIGHT IGNORE	1	1.2	0
5	JLint	1	Overflow	Overflow	1	2	0
6	JLint	1	Domain	Shift count	1	2	0
7	FindBugs	1	CORRECTNESS	ICAST BAD SHIFT AMOUNT	1	2	0
8	JLint	1	Zero result	Zero result	1	24.1	0
9	FindBugs	2	CORRECTNESS	NP UNWRITTEN FIELD	5	28.7	44.90
10	FindBugs	2	STYLE	SA LOCAL SELF ASSIGNMENT	1	47.1	0
115	PMD	3	Basic Rules	Jumbled Incrementer	3	1733	182.59
116	FindBugs	2	MT CORRECTNESS	SC START IN CTOR	2	1735	0
117	FindBugs	2	BAD PRACTICE	DM EXIT	3	1787	60.41
118	FindBugs	2	BAD PRACTICE	ES COMPARING STRINGS WITH EQ	2	1892	0
119	FindBugs	2	MT CORRECTNESS	WA NOT IN LOOP	2	1892	0
120	FindBugs	2	BAD PRACTICE	RR NOT CHECKED	1	1892	0
121	FindBugs	2	MALICIOUS CODE	MS OOI PKGPROTECT	2	1892	0
122	FindBugs	2	BAD PRACTICE	SR NOT CHECKED	1	1892	0
123	FindBugs	1	BAD PRACTICE	SR NOT CHECKED	1	1892	0
124	FindBugs	2	MALICIOUS CODE	EI EXPOSE STATIC REP2	1	1892	0

There are ambiguities in this experiment. For example, suppose *foo.c* changes over transaction 1, 2, 3, and 4 as shown in Figure 3. Suppose a warning instance in the (x) category exists at transaction 1 and another warning instance in the (x) category was added at transaction 2. At transaction 3, a warning instance in the (x) category was removed and at transaction 4, the remaining one was removed.

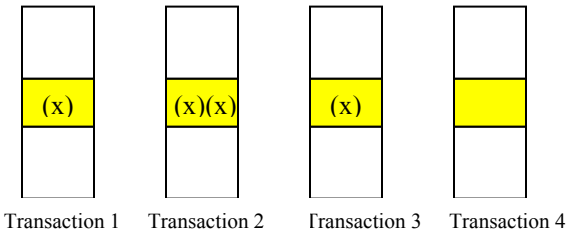

Figure 3. A warning instance addition and removal example of foo.c.

It is not clear whether between transaction 1 and 2, two warning instances are added (and the warning instance in transaction 1 is deleted) or only one warning instance is added (and the warning instance in transaction 1 remains). Similarly, it is not clear which warning instance is removed between transaction 2 and 3. It is possible that all warning instances disappear and a new warning instance (in the same category) appears between transaction 2 and 3. To simplify the experiments, we measure a lifetime as the period of the first appearance of warning instances from a warning category until there is no warning instance from the same category per file. In this example, the lifetime of the warning category is the time difference between transaction 1 to transaction 4.

5. Discussion

It is hard to trace line changes and it is also hard to trace warning instance changes between transactions. The annotation graph could solve this problem partially by mapping each line between two transactions [12]. When a file name changes between two transactions, we will lose warning category change trends or lifetime information, it is necessary to use origin analysis techniques [5, 7]. Even though it is possible that warning instance lifetimes in a category vary, we use the average to measure a category lifetime. Our prioritization approach does not remove false positives of warning instances. However, it puts the bugs (true positives) at the top of the warning list and false positives at the bottom to make bug finding tools more useful.

Our analysis does not utilize developer assessments of the severity of each problem: our technique assumes that the more critical problems are fixed quickly, but it is conceivable that some important problems (say that cause incorrect behavior) are not corrected for a long time.

6. Conclusions and Future Work

We ran three bug finding tools on each development transaction of two open source projects, Columba and JEdit. We computed the lifetime of each warning category, and found that some warning categories have a short lifetime, while others have a long lifetime. We propose prioritization of each warning category based on its observed lifetime. This is a generic prioritization approach applicable to any bug finding tools.

For future work we need to evaluate the prioritized warning categories through user study or validation using the change history.

7. References

[1] C. Artho, "JLint - Find Bugs in Java Programs," 2006, http://jlint.sourceforge.net/.

[2] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey, "Facilitating Software Evolution with Kenyon," Proc. of the 2005 European Software Engineering Conference and 2005 Foundations of Software Engineering (ESEC/FSE 2005), Lisbon, Portugal, pp. 177-186, 2005.

[3] C. Boogerd and L. Moonen, "Prioritizing Software Inspection Results using Static Profiling," Proc. of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'06), pp. 149-160, 2006.

[4] T. Copeland, *PMD Applied*: Centennial Books, 2005.

[5] M. W. Godfrey and L. Zou, "Using Origin Analysis to Detect Merging and Splitting of Source Code Entities," *IEEE Trans. on Software Engineering*, vol. 31, no. 2, pp. 166-181, 2005.

[6] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," Proc. of the 19th Object Oriented Programming Systems Languages and Applications (OOPSLA '04), Vancouver, British Columbia, Canada, pp. 92-106, 2004.

[7] S. Kim, K. Pan, and E. J. Whitehead, Jr., "When Functions Change Their Names: Automatic Detection of Origin Relationships," Proc. of 12th Working Conference on Reverse Engineering (WCRE 2005), Pennsylvania, USA, pp. 143-152, 2005.

[8] S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead, Jr., "Automatic Identification of Bug Introducing Changes," Proc. of 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), Tokyo, Japan, 2006.

[9] T. Kremenek and D. R. Engler, "Z-ranking: Using statistical analysis to counter the impact of static analysis approximations," Proc. of the 10th International Symposium on Static Analysis (SAS 2003), San Diego, CA, USA, pp. 295-315, 2003.

[10] N. Rutar, C. B. Almazan, and J. S. Foster, "A Comparison of Bug Finding Tools for Java," Proc. of 15th IEEE International Symposium on Software Reliability Engineering (ISSRE'04), Saint-Malo, Bretagne, France, pp. 245-256, 2004.

[11] J. Spacco, D. Hovemeyer, and W. Pugh, "Tracking Defect Warnings Across Versions," Proc. of Int'l Workshop on Mining Software Repositories (MSR 2006), Shanghai, China, pp. 133-136, 2006.

[12] T. Zimmermann, S. Kim, A. Zeller, and E. J. Whitehead, Jr., "Mining Version Archives for Co-changed Lines," Proc. of Int'l Workshop on Mining Software Repositories (MSR 2006), Shanghai, China, pp. 72-75, 2006.