Bug Inducing Analysis to Prevent Fault Prone Bug Fixes

Haoyu Yang, Chen Wang, Qingkai Shi, Yang Feng, Zhenyu Chen*
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
*Corresponding author: zychen@software.nju.edu.cn

Abstract

Bug fix is an important and challenging task in software development and maintenance. Bug fix is also a dangerous change, because it might induce new bugs. It is difficult to decide whether a bug fix is safe in practice. In this paper, we conducted an empirical study on bug inducing analysis to discover the types and features of fault prone bug fixes. We use a classical algorithm to track the location of the code changes introducing the bugs. The change types of the codes will be checked by an automatic tool and whether this change is a bug fix change is recorded. We analyze the statistics to find out what types of change are most prone to induce new bugs when they are intended to fix a bug. Finally, some guidelines are provided to help developers prevent such fault prone bug fixes.

Keywords: bug inducing, bug fix, mining software repository, software maintenance

1 Introduction

Bug fix is an important and challenging task for industrial and open source software. Developers may change some code in some specific files to fix bugs. These changes could be various, such as, changing certain lines in an existing function, changing the accessibility of a function from private to public or reorder the parameters of a function, etc. These modifications may bring potential danger into software. They may cause exceptional behavior later. In order to record and fix bugs well, software configuration management system and bug tracking system are used to control the process of constructing software and the flow of bugs respectively. software configuration management could cooperate with bug tracking tools and they indicate which commit fixes a specific bug in the bug tracking system.

It is valuable to automatically identify commits that may induce bugs. It enables developers to quickly and efficiently validate many types of bug fixes. However, it is challenging to find the bug inducing changes accurately. SZZ is introduced to automatically identify bug-introducing commits [11]. This algorithm is improved to provide a process for automatically identifying the bug inducing predecessor lines that are changed in a bug-fixing commit [6]. The procedure of the SZZ algorithm firstly identifies the lines modified or deleted in a bug-fixing commit. It is easy to mine a software repository to find the commit that fixes a bug. We can finish it through examining the log message in each commit and check whether it has the keywords "Bug" and "Fixed" [8] or has symbols of bug reports like "#31245" [2] [4] [11], and then identify the bug-inducing change using annotation graph.

We adopt and improve the SZZ algorithm for our empirical study. We classify bug inducing changes into bug-fix ones or non-bug-fix ones. We analyse the change types of bug inducing changes to investigate the laws in the phenomenons. We can figure out that what kinds of change types to fix a bug are more easily to bring in new bugs later.

The main contributions of this paper are as follows.

- We compare the proportion of change types between bug-inducing change which is bug-fix and bug-inducing change which is not bug-fix to figure out what kinds of change types are more dangerous in bug-fix change.
- We do not only focus on the common changes, but also study special changes appeared in object-oriented programs.
- Ratio of bug-fix bug-inducing changes is counted so that we could know what percentage of bugs are induced by bug fixes.

The rest of paper is organized as follows. Section 2 describes bug inducing analysis method in our study. The experiment design and the result analysis are presented in Section 3. We discuss the practical guidelines and related work in Section 4. The last section is the conclusion and future work.

2 Bug Inducing Analysis

2.1 Change and Bug Fix

Bug fix commits could be identified by examining the commit log messages that whether it includes the keywords like "Bug", "Fixed" or has a specific bug report number corresponding to bug tracking system. Deleted changes and modified changes in bug fix commits are assumed as bug fix changes. Our experiment will use this method to find both bug fix changes and bug inducing changes.

Object-oriented programming languages are widely used in software development and they also present many challenges for software maintenance. There are some useful but risky characteristics, which are difficult to handle since some code changes may cause unexpected and non-local effects [10]. In change impact analysis, a key aspect is to transform source code edits into a set of atomic changes [10]. Some change types come from classical object-oriented change analysis, e.g., **lookupChange** [10] and changes related to variable initializers [9], which are checked manually. Others are distinguished by automatic tool. We combine these change types together, all change types are shown in Table 1. The second column shows that whether we check them manually or not.

Table 1. Atomic Change Type Checked

Type Name	Manually
methodAdded	N
codeChanged	N
codeAdded	N
typeDeclarationAdded	N
innerClassAdded	N
fieldAdded	N
importAdded	N
parameterAdded	N
returnTypeChanged	N
accessChanged	N
constructorAdded	N
throwsAdded	N
parameterTypeChanged	N
variableChanged	N
importSectionAdded	N
parameterNameChanged	N
parameterReordered	N
methodBlockAdded	N
accessAdded	N
modifierChanged	N
lookupChanged	Y
instanceFieldInitializerChanged	Y
staticFieldInitializerChanged	Y
instanceInitializerAdded	Y
emptyInstanceInitializerDeleted	Y
instanceInitializerChanged	Y
staticInitializerAdded	Y
staticInitializerDeleted	Y
staticInitializerChanged	Y

2.2 Bug Inducing

The situation that developers do a lot of bad changes just to fix one tough bug occurs frequently especially when time is limited. Some complicated bugs are hard to fix. Developers may use special ways, which may not be consistent with the original design patterns involved. Obviously these kinds of change types are dangerous. As a consequence, more bugs will be induced and the whole software gets into chaos gradually. In this study, we trace a bug-inducing change and identify whether the change is a bug fix change. These changes are classified into two types as follows.

- **Bug Inducing Change (BIC):** A change that induced a bug and is a bug-fix change itself.
- **Non Bug Inducing Change (NBIC):** A change that induced a bug and is not a bug-fix change itself.

NBIC may be the Addition of new features, enhancement or other changes which are not aimed to fix a bug.

2.3 Fault Prone Analysis

The SZZ algorithm uses CVS annotation feature to trace bug-inducing changes. Developers could match a line which is changed in bug fix changes to its most recent modification so that they finally find where the bug inducing is. However, there may exist some biases judging bug inducing just by the most recent modifications since those modifications are likely to be non-behavior changes, e.g., format changes. The improvement of SZZ [6] removes biases caused by non-behavior changes in order to get a higher accuracy. And annotation feature is replaced with annotation graph at the same time, which is more precise. We use the improvement of SZZ in our empirical study.

SZZ algorithm is used to locate the bug inducing changes based on bug fix changes. When the commit has been located, we use the same method to judge whether the located change is a BIC or NBIC. This step does matter since when we find out that the change is BIC and we can analyze its change types. Some types of changes may be dangerous in a degree. We call them fault prone bug fix changes. Counting change types does let the developers know what kinds of bug fix change types are fault prone while others are

Table 2. Project Information

Project Name	Source	Lineofcode	NumofVersions	Brief Introduction
JEdit	SourceForge	186326	23520	linejEdit is a programmer's text editor written in Java
Protostuff	Googlecode	122277	1677	Protostuff is the stuff that leverages google's protobuf.
Encog	Github	122723	3241	Machine learning framework

relatively safe. Hence developers could fix a bug in the right way since this way prevents another bug from being induced.

3 Empirical Study

We conducted three detailed case studies for further investigate what kinds of change types are more dangerous in BIC than NBIC. These three projects are open source projects implemented by java, detail information of them is shown in the Table 2. As for JEdit[1], there are too many revisions for us to examine all of them, so we choose to select revisions randomly. For Protostuff[2] and Encog[3], we investigate the whole revisions of them.

3.1 Experiment Procedure

Bug-fix commits are identified by their log message. Deleted and modified changes are being tracked backward using annotation graph so that we can find out where the corresponding bug-inducing change is. Then we judge whether the bug-inducing change is a BIC with the same method. Some bugs are induced in ordinary commit or initial import. They are caused by NBIC, while others are caused by BIC, which we count for percentage. After we locate the bug-inducing change, we use a tool *diffJ*[4] to judge what atomic change types it includes. However, some atomic change types from object-oriented perspectives cannot be distinguished by automatic tool. So we manually check them.

3.2 Result Analysis

The statistics are shown in Table 3. The data tell us that in each project, there exist some BIC which lead to more bugs later.

Then we separate BIC from NBIC, comparing the differences between the proportion of each atomic change type in BIC and NBIC respectively. The results are shown in Figure 1, 2 and 3 in descending order. It could be concluded that

[1]http://sourceforge.net/projects/jedit/
[2]https://code.google.com/p/protostuff/
[3]https://github.com/encog/encog-java-core
[4]https://github.com/jpace/diffj

Table 3. Overall Experimental Results

Project Name	NBIC(#)	BIC(#)	BIC(%)
JEdit	466	144	23.61%
Protostuff	180	71	28.29%
Encog	298	39	11.57%

codeAdded and **codeChanged** appeared in BIC are much more than NBIC, which means that these change types are more prone to cause other bugs later if they appeared in BIC.

To discover more detailed information, we conduct further investigation on the atomic change types **codeAdded** and **codeChanged** since they appear more in BIC than NBIC. In that case, we manually check the BIC which includes atomic change type **codeAdded** or **codeChanged** in order to conclude, in which context these change types will show up. For example, adding or changing an *if* conditional statement, adding or modifying the parameters of the method, removing or changing the value of some local variables. Our statistics are shown in Table 4.We list the percentage of all the change types they related to. Some minor changes are combined together, for example, adding and modifying the parameters of method are combined to parameters. The total number of **codeAdded** and **codeChanged** , method invoke changes and return value are listed as *total*,*method* and *return* respectively.

3.3 Threat to Validity

One threat to external validity is mainly due to the quality of open source project. Well managed project will do good to our research, while bad one will make biases. As for our experiment, the project *encog* initially imports 605 files together into repository, which leads to a lot of bugs tracked backward to the first revision. Obviously the first version is not a bug-fix commit. As a result, percentage of BIC is relatively small compared to the other projects.

Furthermore, we investigate only three projects and all of them are open source projects, which may not be enough to support our conclusion. Our experiment results may not be representative, because there may exist great difference in design between open source software and industrial software.

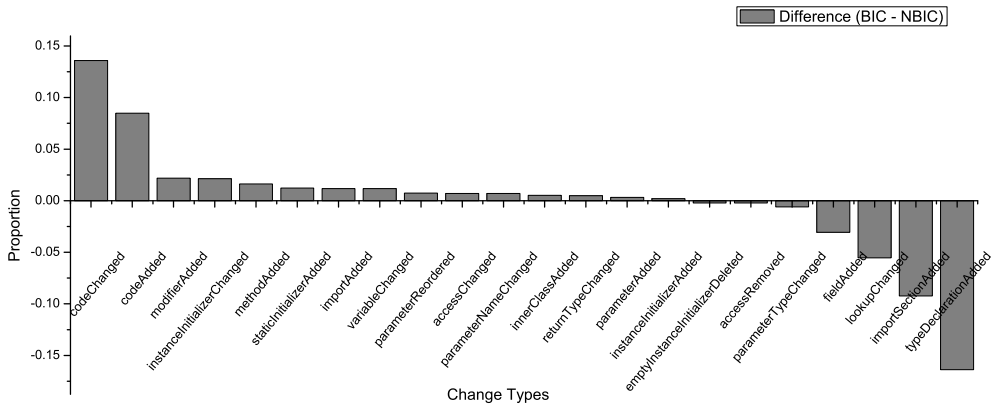

Figure 1. Proporation Difference of Each Change Type between BIC and NBIC in JEdit

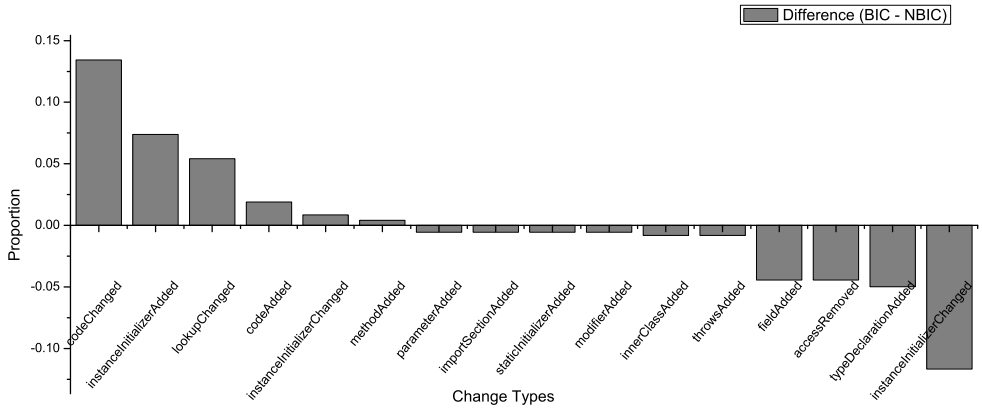

Figure 2. Proporation Difference of Each Change Type between BIC and NBIC in Protostuff

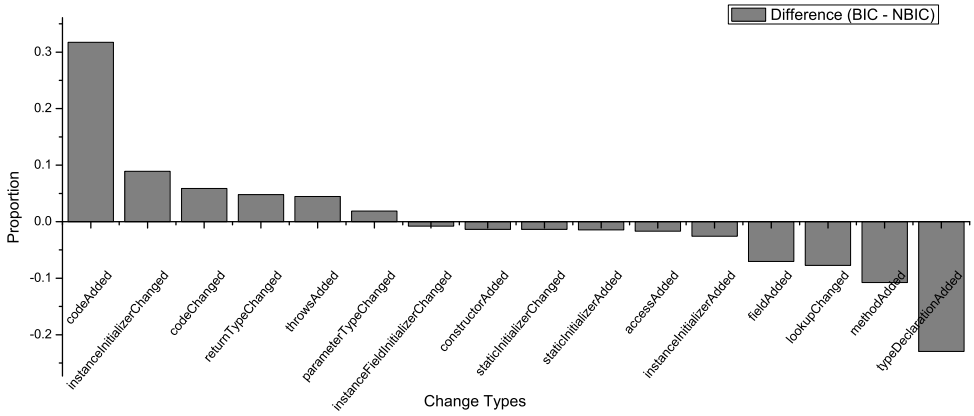

Figure 3. Proporation Difference of Each Change Type between BIC and NBIC in Encog

Table 4. Detail statistics about codeAdded and codeChanged changes

Project	total	if	method	local variable	return	type cast	parameters	file path	loop	exception
JEdit	68	42.6%	8.8%	23.5%	2.9%	1.5%	7.4%	1.5%	4.4%	2.9%
Protostuff	26	69.2%	15.4%	3.8%	3.8%	0	7.7%	0	0	0
Encog	21	38.1%	23.8%	9.5%	0	0	19%	0	9.5%	0
Overall	115	47.8%	13.4%	16.5%	2.6%	0.86%	9.5%	0.86%	4.3%	1.7%

Another threat is that all the three projects conducted are implemented by Java. Although Java is a representative object-oriented language, other languages may include some special features which java language does not include, and we do not take them into consideration.

We manually check some kinds of change types in object-oriented programs. However, manual checks may include errors.

4 Discussion

In this section, some interesting findings from our statistics will be discussed, in order to provide some guidelines in the software development and debugging practice.

4.1 If-Else Clauses Are Dangerous

As shown in the Table 4, changes related to *if* conditional statement occupy a large proportion of both **codeChanged** and **codeAdded**.

These changes include adding some new if-else blocks between the lines of a method, making some modifications of the conditional statements and applying combination or separation on the original statements. According to our statistics, the proportion of these changes on *if* conditional statements accounts for nearly 50% in both **codeChanged** and **codeAdded** changes, while other kinds of change types are in minority, which reveals that it should not be conceived an ignorable issue during the process of bug-fix. All of these changes on *if* conditional statements cast great danger on the lines just added or modified. Chances are that the conditional statements run normally under current circumstances, however, the coverage of the condition may not be integrate.

As the percentage of changes related to *if* conditional statements is so high in the bug-prone changes in our experiment, we sincerely suggest the programmers to avoid these kinds of changes in their software development practice. One effective solution is to apply widely recognized software design patterns and strict object-oriented rules on the program during the beginning period of the project in order to decrease the modifications on initial codes of a class or method. For example, using strategy pattern would decrease the number of *if* conditional statements in one method. Thus, the complexity of the business logic of a class and the possibility of changing the conditional statements to fix a bug will be reduced.

4.2 Open/Closed Principle

From the statistics, we can see that the percentage of **typeDeclarationAdded** in BIC is much lower than N-BIC. Thus, we can make the assumption that **typeDeclarationAdded** may be a much safer way to fix a bug. There is a famous principle in object-oriented software development supporting our idea, and that is the so-called **Open/Closed Principle**.

The **Open/Closed Principle** states that *"software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"*. [7]. That means if we want to add some new functions or fix the existed bugs, such entities should allow these behaviours without altering its source code. If a project is constructed under the **Open/-Closed Principle** strictly, there will be an increasing trend in BIC which can be completed by declaring new types or extending the current interface but not modifying them. As a result, changes like **codeChanged** or **codeAdded** which are prone to induce new bugs in BIC will be greatly prevented.

4.3 Related Work

As concluded in the work of Mockus [8], Cubranic [2] and Fischer [4], commit in software configuration management system can be judged whether it is a bug-fix one by the log message. Some key words, e.g., "Bug" or "Fixed" even bug report number "#31245" can serve as symbols of a bug-fix commit. Based on their work, Sliwerski et al. [11] presented SZZ algorithm to find out the position of bug-inducing changes according to the position of bug-fix changes. Kim et al. [6] improved SZZ algorithm by using annotation graph and ignore the changes which is meaningless just like blank line changes and format changes so that it became much more precise. The improvement of SZZ is currently the best available algorithm for automatically identifying bug-inducing commits. The algorithm does great contributions to related researches. Kim et al. [5] evaluate

whether a change is risky by using SZZ to track the original change. D'Ambros [3] visually reveals the relationship between bugs and software evolution. Williams et al. [13] revisited SZZ algorithm, using SZZ algorithm to track bug-inducing changes and identify change types of them, which is similar to our work. Ryder et al. [10] analyse the impact of object-oriented changes. Ren [9] does the similar work as Ryder, which defined some object-oriented change types that we use in our paper. Spacco [12] used lining mapping which make the tracking of bug inducing fix more precisely. A differencing technique and tool for object-oriented programs: JDiff [1] provide us the tool support. [12]

5 Conclusion and Future Work

In this paper, we apply the SZZ algorithm on a series of projects to find out what kinds of bug-inducing changes exist higher possibilities to become a great threat when they are marked as bug-fix changes. Based on the empirical studies operated on three open source projects, our statistics show that the **codeAdded** and **codeChanged** changes, especially addition or modification of the *if* conditional statements in bug-fix commits are apt to cause more bugs in the future.

Our future work may include the following aspects: A much wider selection of projects are going to be researched to further validate the conclusion we drew. Other change types except for **codeAdded** and **codeChanged** may also reveal some regular patterns as the number of projects grows.

Acknowledgment

The work described in this article was partially supported by the National Basic Research Program of China (973 Program 2014CB340702), the National Natural Science Foundation of China (No. 61373013, 61170067).

References

[1] T. Apiwattanapong, A. Orso, and M. J. Harrold. Jdiff: A differencing technique and tool for object-oriented programs. *Automated Software Engineering*, 14(1):3–36, 2007.

[2] D. Cubranic and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the International Conference on Software Engineering*, pages 408–418, 2003.

[3] M. D'Ambros and M. Lanza. Software bugs and evolution: A visual approach to uncover their relationship. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering*, pages 10–238, 2006.

[4] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance*, pages 23–32, 2003.

[5] S. Kim, E. J. Whitehead, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.

[6] S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead. Automatic identification of bug-introducing changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 81–90, 2006.

[7] B. Meyer. Object-oriented software construction. *Prentice Hall International Series in Computer Science*, 1988.

[8] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance,*, pages 120–130, 2000.

[9] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. 39(10):432–448, 2004.

[10] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 46–53, 2001.

[11] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.

[12] J. Spacco, D. Hovemeyer, and W. Pugh. Tracking defect warnings across versions. In *Proceedings of the 2006 international Workshop on Mining software repositories*, pages 133–136, 2006.

[13] C. Williams and J. Spacco. Szz revisited: verifying when changes induce fixes. In *Proceedings of the 2008 workshop on Defects in large software systems*, pages 32–36, 2008.