Mining Frequent Bug-Fix Code Changes

Haidar Osman, Mircea Lungu, Oscar Nierstrasz
Software Composition Group
University of Bern
Bern, Switzerland
{osman, lungu, oscar}@iam.unibe.ch

***Abstract*—Detecting bugs as early as possible plays an important role in ensuring software quality before shipping. We argue that mining previous bug fixes can produce good knowledge about why bugs happen and how they are fixed.**

In this paper, we mine the change history of 717 open source projects to extract bug-fix patterns. We also manually inspect many of the bugs we found to get insights into the contexts and reasons behind those bugs. For instance, we found out that missing null checks and missing initializations are very recurrent and we believe that they can be automatically detected and fixed.

I. Introduction

Software evolution is inevitable. In this process, source code changes many times in correspondence to changing requirements, improvements, and bug fixes. Bug-fix changes are important to study as a starting point to understand bugs, predict them, and ultimately fix them automatically.

Bug prediction is a hot topic in research. Many approaches try to predict bugs based on code metrics [1][2] (lines of code, complexity, *etc.*), or on process metrics [3][4] (number of changes, recent activities, *etc.*), or previous defects [5]. Those predictors do not point out the bugs themselves but rather predict the possibility of having a bug in a certain file or module.

Usually, the research on change patterns is handled first by manual categorization of change patterns and then counting automatically the instances of the identified patterns. We argue that the process should be the other way around.

In this paper we automatically analyze the change history of 717 open source projects, parse them to find the buggy code and the fix code, and extract the change patterns that correspond to bug fixes. After that we go deeper into the rabbit hole by manually reviewing the most important and recurring fix patterns to find out the reasons and contexts of those patterns.

Our motivation for this work is that we have a tremendous amount of data in the change history of numerous software projects. This change history contains countless bug fixes of all types and sizes. We believe that mining these bug fixes in a systematic way can provide deep insights on how bugs are created and fixed. We believe that about 40% of the bugs are very frequent that we can locate and fix them automatically.

In summary, this paper makes the following three contributions:

1) We demonstrate a new approach to automatically extract bug fix patterns.
2) We present some interesting statistical findings about the bugs and fixes in general.
3) We explore and discuss the fourteen most recurring patterns and their reasons and contexts.

In our study we learn that missing null checks are very recurrent and they usually cause null-pointer exceptions that lead to crashes. We also find out that there are some type of objects that need special initialization or configuration immediately after instantiation. Another interesting finding is that bad naming of variables and methods can lead to serious bugs.

II. Procedure

The procedure we followed to get our bug and fix patterns can be divided into three separate phases: building the software corpora, analyzing the source code, and manual inspection.

A. Collecting Java Projects

The first phase is to gather the right software corpora. We cloned 717 Java projects from GitHub with the aid of a tool that queries GitHub for the biggest and most popular Java projects and clones them locally.

B. Analyzing Source Code and Change History

In the second phase, we analyse the change history and source code of the cloned projects. Figure 1 illustrates the main steps we perform for every project in our corpus.

a) Find Bug-Fix Commits: We parse the change history of the project and extract the revisions that correspond to bug fixes[1]. In other words, every commit corresponds to a bug fix if the commit message contains the keyword "fix" or its derivations.

b) Extract Bug Fix: We extract two versions of each changed method: the version before the fix (buggy method), and the version after the fix (fixed method)[2].

c) Compare: We compare the two version of the methods and extract exactly the changed code. We call the minimal code that had the bug a *bug hunk*, and we call the minimal code that fixed the bug a *fix hunk*, as in Figure 2. Then every bug hunk and fix hunk are concatenated with the string

[1] We used the JGit library for parsing GIT history changes. http://www.eclipse.org/jgit/

[2] We used the JAPA library for parsing Java code. http://code.google.com/p/javaparser/

978-1-4799-3752-3/14/$31.00 © 2014 IEEE

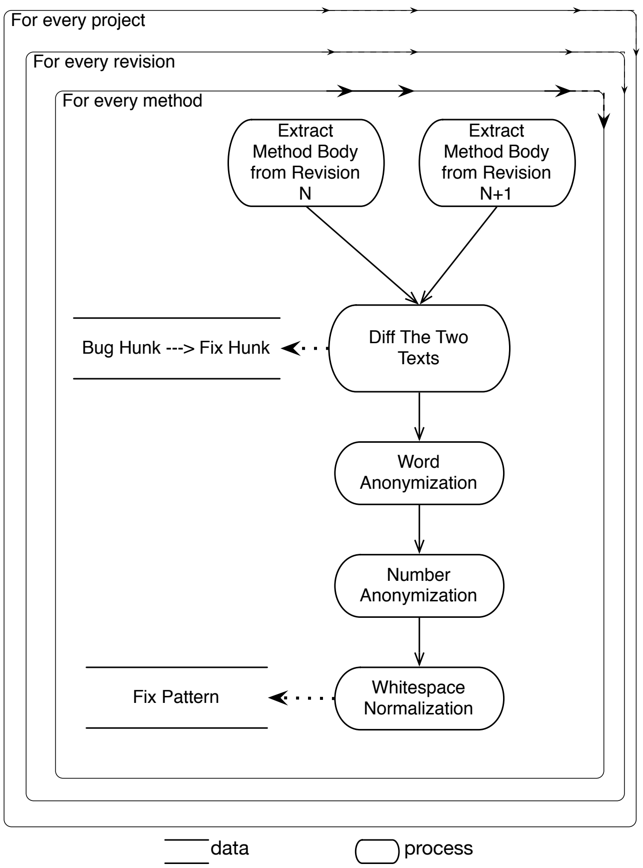

Fig. 1. The pipeline through which each project has to go through. The final output of this analysis is the fix patterns and their concrete instances in the software projects.

“>>>> ” in between to form a code transition layout, *e.g.*, *callback.handle(row); >>>> callback.apply(row);*

d) Normalize: We anonymize and normalize the patterns by applying the following three steps:

- Each word (variable name, method name, *etc.*) is replaced by the letter “T” (except for java keywords like *return, int, null,* etc.).
- Each number is replaced by “0”.
- Each sequence of white spaces (spaces, tabs, newlines, *etc.*) is replaced by one space.

In this way, we end up with anonymized patterns like:
T.T(T); >>>> T.T(T);

C. Manual Inspection

In the third phase and after gathering all the necessary information, we grouped the code changes by the fix patterns. So for each change pattern, we have the number of occurrences, the number of projects where the pattern occurred, and the concrete code snippets that represent the pattern.

Then we ordered the fix patterns based on their distributions across the projects. In other words, the more the pattern appears in different projects, the more significant it is.

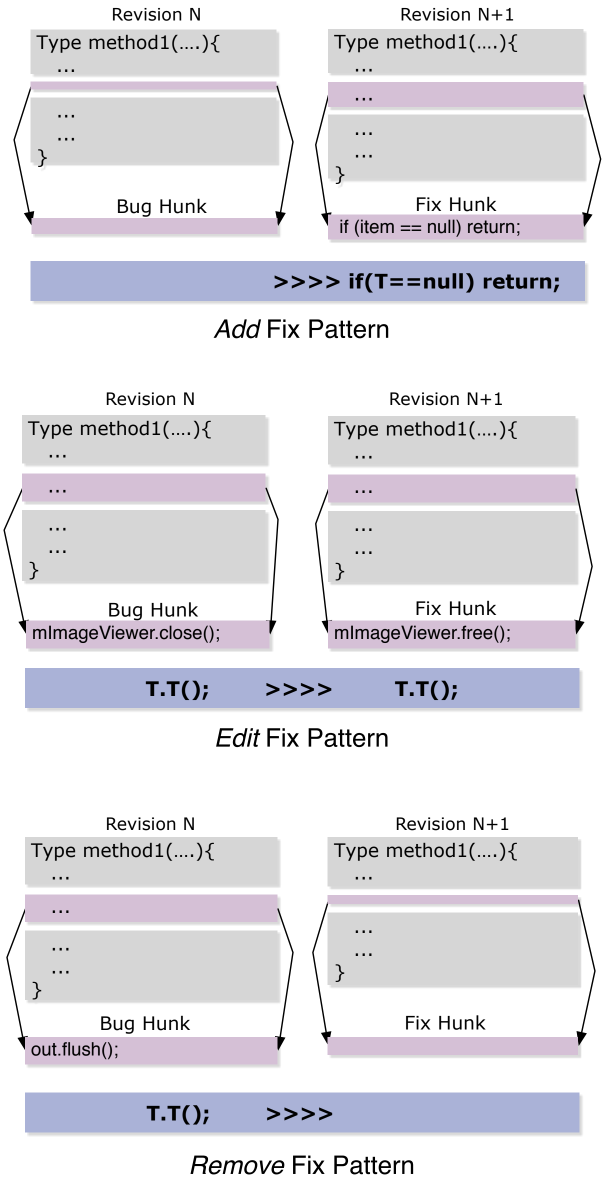

Fig. 2. This diagram contains the three types of code changes corresponding to bug fixes and how they are represented as patterns.

Then we started inspecting the instances of each of the most significant patterns by reading the actual buggy code, how it was fixed, commit messages, and bug reports when possible. For each of the patterns reported in this paper, we manually inspected at least a hundred randomly-selected commits thoroughly.

III. Findings & Results

Our analysis includes 717 Java software projects and 190,821 code changes corresponding to 94,534 bug-fix commits in total. During our investigation on the bug fixes, we found out that about 53% of the fixes involve only one line of code, as in figure Figure 3, and most of bug fixes (73%) are less than four lines of code.

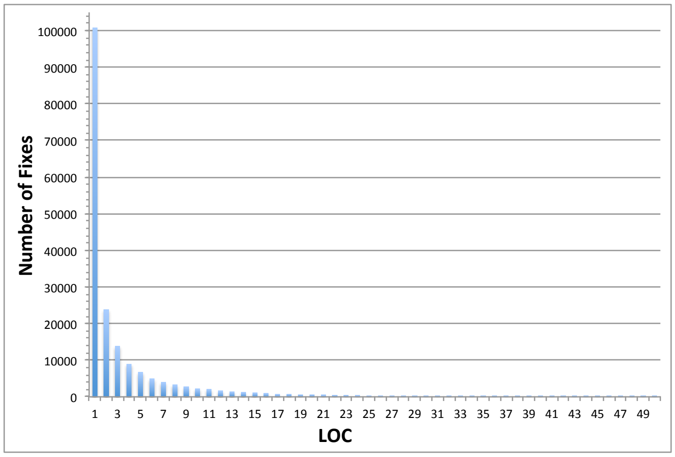

Fig. 3. The distribution of bug fixes according to the number of changed lines of code.

Also we found out that about 40% of the bug-fixing code changes are recurrent patterns that appear in more than 10 projects as demonstrated in Figure 4. This leads us to the conclusion that the fixing of those patterns can be automated and very useful for many projects.

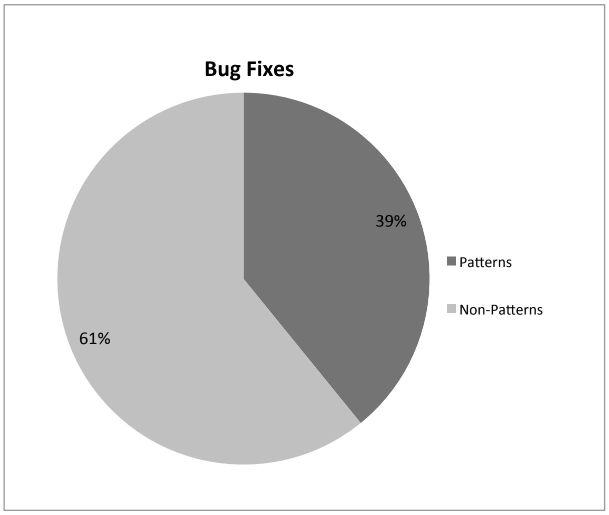

Fig. 4. This piechart shows that 39% of the bug fixes are frequent patterns. A bug-fix change is considered frequent if it appears in more than 10 projects.

Due to the large number and diversity of the analyzed projects, the most recurrent patterns are high-level and have nothing to do with the business logic of those projects. In the following subsections, we categorize the most frequent patterns according to the reasons behind them.

A. Missing Null Checks

The fixes for the bugs in this category are all *addition* changes. They involve the addition of a null check on a certain object (*the checked object*) like *if (T==null)* or *if (T!=null)*. Table I shows the fix patterns that fall in this category. This type of bugs appears in 48% of the examined projects making it a serious and very frequent problem.

TABLE I
THE BUG-FIX PATTERNS OF THE MISSING NULL CHECK. MORE THAN 48% OF THE PROJECTS ARE INVOLVED

	Instances	Projects
␣ ⟶ *if(T != null)*	3,718	316
␣ ⟶ *if(T == null) return;*	1051	190
␣ ⟶ *if(T == null) return null;*	243	80
␣ ⟶ *if(T == null) throw new T();*	207	75
␣ ⟶ *if(T == null) T=T;*	157	67
␣ ⟶ *if(T == null) continue;*	82	34
Total	5,172	348

In many of the cases, a blank *return* will follow the check. In other words, if the checked object is null then the method cannot continue its execution and should immediately return. One example of this is:

```
if (viewActionsContainer == null) return;
```

The checked objects are either the results of method invocations, parameters, or member variables.

In almost 70% of the cases the checked object comes from a method invocation. Moreover, this kind of bug often appears when chaining method calls as the examples in Figure 5 show.

Bug Hunk
```
((View) getParent()).invalidate();
```

Fix Hunk
```
View parent = (view) getParent();
if(parent!=null)
   parent.invalidate();
```

Bug Hunk
```
((SimpleAdapter)getListAdapter())
    .notifyDataSetChanged();
```

Fix Hunk
```
SimpleAdapter adapter =
  (SimpleAdapter)getListAdapter();

if (adapter != null)
  adapter.notifyDataSetChanged();
```

Fig. 5. Null-pointer exceptions due to missing checks

B. Missing Invocation

The fix patterns for this category of bugs are also *addition* changes. Table II shows the patterns in this category.

TABLE II
THE BUG-FIX PATTERNS OF THE MISSING INVOCATION.

	Instances	Projects
␣ ⟶ *T.T();*	995	188
␣ ⟶ *T.T(T);*	747	165
␣ ⟶ *T();*	667	140
Total	2,409	288

There are many scenarios in which this kind of bug can appear. The first is a **missing initialization or configuration of an object immediately after its creation**. For example:

```
ConnectionPool config=new ConnectionPool()
```

should be immediately followed by:

```
config.initialize()
```

The second scenario is what we can call a **missing refresh** where before doing something or after finishing something, a certain object should be brought to a consistent state or "*refreshed*". These missing method invocations are either at the very beginning or at the very end of a method body. Method names that we often encountered from this category are: *refresh, reset, clear, pack, repaint, redraw,* etc. In most of the cases, the "*refreshed*" object is of some kind of a container or an aggregator class like canvas, tree, view, *etc.*.

The third scenario is when there is a **missing release**. This type of invocation is always about freeing resources and always comes at the end of the method body. Example methods are: *release, delete, dispose, close,* etc.

C. Wrong Name

The fix patterns for this category of bugs are *edit* changes. They are shown in Table III.

TABLE III
THE BUG-FIX PATTERNS OF THE NAMING PROBLEMS CATEGORY.

	Instances	Projects
return T; ⟶ *return T;*	305	110
T.T(T); ⟶ *T.T(T);*	329	78
T.T(); ⟶ *T.T();*	90	35
Total	724	161

The bug lies in the object names, method names, or parameter names. The reasons behind this kind of bug are either (1) using the wrong identifier due to name similarity or (2) calling the wrong method based on mistaken name-driven assumptions about its functionality.

Figure 6 shows two fixes were the bugs were due to misunderstanding method names.

```
imageViewer.close(); ⟶
imageViewer.freeTextures();

key.rewind(); ⟶
key.flip();
```

Fig. 6. Sometimes developers invoke wrong methods due to misunderstanding method names.

Figure 7 shows that the names can be very similar and the programmer might mistakenly use one instead of another.

```
dragView.setPaint(mPaint);⟶
dragView.setPaint(mHoverPaint);

visitedURLs.clear(); ⟶
visitedURIs.clear();

return hasClassAnnotations; ⟶
return hasRuntimeAnnotations;
```

Fig. 7. In some cases developers confuse objects or parameters with similar names.

D. Undue Invocation

The fix patterns for this category of bugs are *remove* changes. The only pattern we found is {*T();* ⟶ ␣ } which occurred 186 times in 70 projects. This pattern is exactly the opposite of the missing invocation pattern and corresponds to the same kind of methods like *flush, reindex, init, close,* etc..

The methods in this category are resetters, initializers, or resource releasers. So the bug comes either from premature resource freeing or resetting, or unnecessary initialization.

IV. Threats to Validity

A. Objects

When we selected the projects in our corpora, our search criteria submitted to GitHub specified projects that were last updated at least on 01.01.2013, had more than a five-star rating, and were more than 100KB in size. Basically these search criteria ensure that those projects are still maintained, popular, and big enough to be a good representative sample of the open source Java projects.

B. Methodology

In our parsing, anonymization, and normalization of the bug-fix code changes, we might have missed some instances or grouped some instances in the wrong categories. But due to our large number of code change snippets (190821 code changes), we argue that the exceptional cases will affect our statistics and pattern significance computation within acceptable limits. Also it should be noted that the whole process of automatic pattern categorization serves only to guide our research and investigation on software bugs and not to claim any significant statistical results.

C. Subjects

The categorization and the reasoning about the bug fixes were done by the first author through manual inspection. We are aware that this might lead to some subjective bias. So the second author verified and manually inspected 5% of the bug fix samples chosen randomly. Both categorizations were completely consistent. Nevertheless, we are not claiming any statistical significance and again the whole study will be evaluated against the real automatic fix approach to come.

V. Related Work

There is extensive research in the field of change patterns in general and fix categorization in specific. Pan *et al.* [6] manually found some change patterns that correspond to fixes and automatically extracted instances of those patterns from seven open source projects. The patterns they extracted are of a high-level abstraction like *Addition of Precondition Check* or *Different Method Call to a Class Instance*.

Kim *et al.* created a tool named BugMem [7] that extracts bug fix rules from the history of a project and applies bug detection and fix suggestion based on that past. This approach is smart and innovative but the rules are not "patterned" and they are instead saved in a concrete form. This leads to the saved fix rules being applicable only to code clones within the same project. In this case, code clone tracking tools would perform definitely better by following the changes of a clone and applying it on all other clones.

Livshits and Zimmermann developed a tool called DynaMine [8] that finds recurrent patterns of application-specific method invocations. Based on the bias of those patterns, DynaMine can suggest bug locations.

Martinez *et al.* [9] came up with a new approach for specifying and finding change patterns based on AST differencing. They evaluated their approach by counting 18 change patterns from [6] on 6 open-source projects. The granularity of the change patterns are exactly the same as in [6]. On the other hand, Fluri *et al.* [10] used hierarchical clustering and tree differencing to find change pattern but with a very coarse-grained changes like *development change* and *maintenance change*.

Our work is different from the aforementioned approaches in three ways:

1) Our study included a large number of software projects (717 projects) while all the previous approaches involved 7 projects at maximum. This leads to better and more generalizable results.
2) The sequence of our study was to automatically extract patterns first, and then to manually revise those patterns. This human analysis phase was crucial to get insights into the reasons behind the bugs.
3) Our change granularity is at the level of single lines of code. So all changes are captured without any abstraction. For example, in our approach, a method call with two parameters is different from a method call with one parameter. Whereas in the AST differencing, both are considered as method calls.

VI. Future Work

Since there is a large number of diverse Java projects in the studied corpora, the patterns we found are high-level language-related patterns. But still we noticed that many of the bugs actually come from wrong usage of external libraries or frameworks, especially the bugs in missing invocation pattern. In any case, we think that further analysis of library usages should be undergone to reveal framework-related patterns.

Also we noticed a significant keyword similarity within each of the aforementioned categories. We believe that text analysis of the source code can reveal some more precise bug-fix patterns.

We intend to use the collected knowledge to predict bug exact locations in the source code and suggest the proper fixes at compile time.

References

[1] N. Nagappan and T. Ball, “Static analysis tools as early indicators of pre-release defect density,” in *Proceedings of the 27th international conference on Software engineering*, ICSE ’05, (New York, NY, USA), pp. 580–586, ACM, 2005.

[2] N. Nagappan, T. Ball, and A. Zeller, “Mining metrics to predict component failures,” in *Proceedings of the 28th international conference on Software engineering*, ICSE ’06, (New York, NY, USA), pp. 452–461, ACM, 2006.

[3] A. E. Hassan, “Predicting faults using the complexity of code changes,” in *Proceedings of the 31st International Conference on Software Engineering*, ICSE ’09, (Washington, DC, USA), pp. 78–88, IEEE Computer Society, 2009.

[4] A. Bernstein, J. Ekanayake, and M. Pinzger, “Improving defect prediction using temporal features and non linear models,” in *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, IWPSE ’07, (New York, NY, USA), pp. 11–18, ACM, 2007.

[5] S. Kim, T. Zimmermann, E. J. W. Jr., and A. Zeller, “Predicting faults from cached history,” in *ICSE ’07: Proceedings of the 29th international conference on Software Engineering*, (Washington, DC, USA), pp. 489–498, IEEE Computer Society, 2007.

[6] K. Pan, S. Kim, and E. J. Whitehead, Jr., “Toward an understanding of bug fix patterns,” *Empirical Softw. Engg.*, vol. 14, pp. 286–315, June 2009.

[7] S. Kim, K. Pan, and E. E. J. Whitehead, Jr., “Memories of bug fixes,” in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT ’06/FSE-14, (New York, NY, USA), pp. 35–45, ACM, 2006.

[8] B. Livshits and T. Zimmermann, “Dynamine: finding common error patterns by mining software revision histories,” *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 296–305, Sept. 2005.

[9] M. Martinez, L. Duchien, and M. Monperrus, “Automatically extracting instances of code change patterns with ast analysis,” in *Proceedings of the 29th IEEE International Conference on Software Maintenance*, 2013. ERA Track.

[10] B. Fluri, E. Giger, and H. Gall, “Discovering patterns of change types,” in *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pp. 463–466, 2008.