Mining Historical Information to Study Bug Fixes

Eduardo C. Campos
Faculty of Computing
Federal University of Uberlândia, Brazil
eduardocunha11@gmail.com

Marcelo A. Maia
Faculty of Computing
Federal University of Uberlândia, Brazil
marcelo.maia@ufu.br

***Abstract*—Software is present in almost all economic activity, and is boosting economic growth from many perspectives. At the same time, like any other man-made artifacts, software suffers from various bugs which lead to incorrect results, deadlocks, or even crashes of the entire system. Several approaches have been proposed to aid debugging. An interesting recent research direction is *automatic program repair*, which achieves promising results towards the reduction of costs associated with defect repair in software maintenance. The identification of common bug fix patterns is important to generate program patches automatically. In this paper, we conduct an empirical study with more than 4 million bug fixing commits distributed among 101,471 Java projects hosted on GitHub. We used a domain-specific programming language called *Boa* to analyze ultra-large-scale data efficiently. With *Boa's* support, we automatically detect the prevalence of the 5 most common bug fix patterns (identified in the work of Pan *et al.*) in those bug fixing commits.**

***Keywords*—Software bugs, Automatic error repair, Bug fix patterns, Human-like patches, Software fault**

I. Introduction

There are more bugs in real-world programs than human programmers can realistically address [4]. The battle against software bugs exists since software existed. It requires much effort to fix bugs, e.g., Kim and Whitehead [3] report that the median time for fixing a single bug is about 200 days. Program evolution and repair are major components of software maintenance, which consumes a daunting fraction of the total cost of software production. Research in *automatic program repair* has focused on reducing defect repair costs and are therefore especially beneficial. Moreover, research in this direction has already produced promising results. For example, Le Goues *et al.* [4] reported that their approach was able to automatically fix 55 out of 105 bugs. However, the research community has limited knowledge on the nature of bug fixes [7] and still does not have general consensus on which kinds of software bugs are most common [8].

This paper presents an in-depth investigation on the bug fixing commits of Java programs, taken from several million human-made bug fixes from GitHub (the world's largest collection of open-source software, with more than 23 million public repositories). This software repository contains an enormous collection of software and information about software [1]. We used a domain-specific programming language called *Boa* [1] to analyze ultra-large-scale data efficiently. In particular, we analyzed the characteristics of those bug fixing commits from different perspectives in order to answer the four research questions below:

$\mathbf{RQ_1}$: *Considering bug fixing commits associated with only one file, which file types are usually changed to fix a bug?*

$\mathbf{RQ_2}$: *Which kinds of statements appear more frequently in bug fixing commits?*

$\mathbf{RQ_3}$: *What is the prevalence of the 5 most common bug fix patterns identified in the work of Pan et al. in the analyzed bug fixing commits?*

$\mathbf{RQ_4}$: *How many distinct developers committed these bug fixing commits? And how many committers are there for each analyzed Java project?*

To sum up, our contributions are as follows:

- We have shown that developers often forget to add `IF` preconditions in the code. One proof of this is that the bug fix pattern that most appeared in the analyzed bug fixing commits of *Boa* dataset was IF-APC (Addition of IF Precondition Check);
- Our findings suggest that mutation based program repair may need to consider multi-language programming and bugs in non-source files (e.g., configuration files). Our results confirm the findings obtained by Zhong and Su [11] (please see the Findings 1, 2, 13, and 14 for more details). This is relevant because current automatic repair approaches have been evaluated on only source files belonging to a limited number of programming languages (such as C and Java).

II. Related Work

Zhong and Su [11] designed and developed *BugStat*, a tool that extracts and analyzes bug fixes. They conducted an empirical study on more than 9,000 real-world bug fixes from six popular Java projects. Their results provide useful guidance and insights for improving the state-of-the-art of automatic program repair. We study a much larger dataset [1] with 101,471 Java projects. Moreover, we designed *Boa* programs that automatically detect the five most common bug fix patterns identified in the work of Pan *et al.* [8]. Martinez and Monperrus [6] analyzed the links between the nature of bug fixes and automatic program repair. Furthermore, the empirical study focuses on only one aspect of automatic program repair, namely the search space of fixing bugs. They mined repair models from manual fixes, and the mined repair models improve random search. Our study provides findings to better understand and improve these approaches. For example, we notice that many bugs reside in source files of different programming languages or in non-source files (e.g., configuration files). Xuan *et al.* [10] proposed *Nopol*, an approach to automatic repair of buggy conditional statements (e.g., `if-then-else` statements). In our dataset, the bug fix pattern that appears more frequently is IF-APC (*Addition of IF Precondition Check*), totaling 29.2019% of the analyzed bug fixing commits. Kim *et al.* [2] proposed an approach called PAR to fix bugs in Java code automatically. PAR is based on repair templates: each of the PAR's ten repair templates represents a common way to fix a common kind of bug. Soto *et al.* [9] conducted a large-scale study of bug fixing commits in Java projects. Their findings provide useful insights for automatic program repair tools in Java. They created *Boa* programs to detect the PAR's bug fix patterns [2] and provided an informative approximation of their prevalence in the *Boa* dataset. We used the same dataset in our study but we created *Boa* programs to detect the five most common bug fix patterns identified in the work of Pan *et al.* [8]. Moreover, we do not limit our study to bug fix patterns. We also investigated other aspects related to human-made bug fixes such as the kinds of statements that appear more frequently in bug fixing commits and the kinds of files that are usually changed to fix a bug.

III. Dataset and Characteristics

In this paper, we use the *September 2015/GitHub* dataset offered by *Boa* [1], including 554,864 Java projects with 23,226,512 commits. *Boa* identifies 4,590,405 as bug fixing commits distributed among 101,471 Java projects (18.2875%). In other words, 81.7125% of Java projects present in this dataset do not have any bug fixing commit. In this paper, we focus our analysis on these 101,471 Java projects because our goal is to study bug fixes and patterns. Figure 1 shows a query written in *Boa* language that returns the number of bug fixing commits in Java GitHub projects. The built-in function `isfixingrevision` (line 6) uses a list of regular expressions to match against the revision's log (i.e., commit's log message). If there is a match, then the function returns true indicating the log most likely was for a commit fixing a bug.

Figure 2 shows the distribution of bug fixing commits among 101,471 Java projects. As we can see in this figure, 81% of these projects have 1 to 15 bug fixing commits. Only 9% of these projects have 51 or more bug fixing commits. This pie chart shows that is not common to see open-source Java projects hosted on GitHub with a large number of bug fixing commits (e.g., more than 50 bug fixing commits).

Programming Language: As returned by *Boa*, the major language of a project is the one with the highest percentage of source code, considering the files in the project. Figure 3 shows the distribution of the analyzed projects per number of programming languages. As we can see, 56,414 out of 101,471 Java projects (i.e., 55.5961%) use only one programming language (i.e., Java). However, 10,837 out of 101,471 Java projects (i.e., 10.6798%) use five or more programming languages.

Kinds of changed files: Table I shows the kinds and descriptions of changed files present in the *Boa* dataset and the number of changed files per file kind. We consider a file *changed* if it is new, modified, or deleted in a commit. In total, 52,052,571 files were changed.

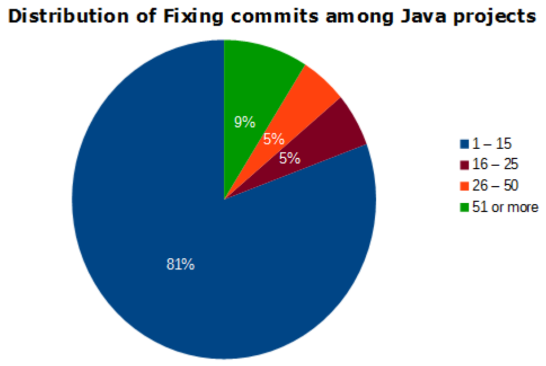


Fig. 2. Distribution of bug fixing commits among Java GitHub projects.

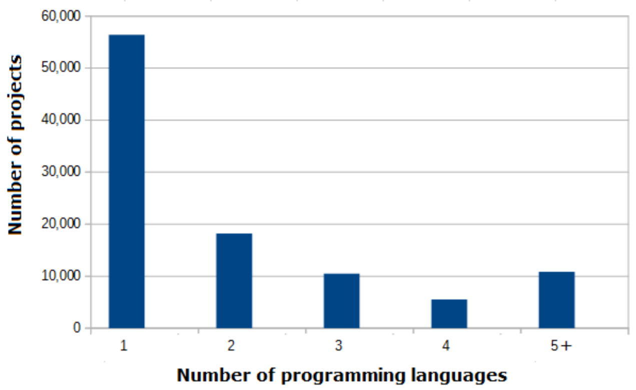


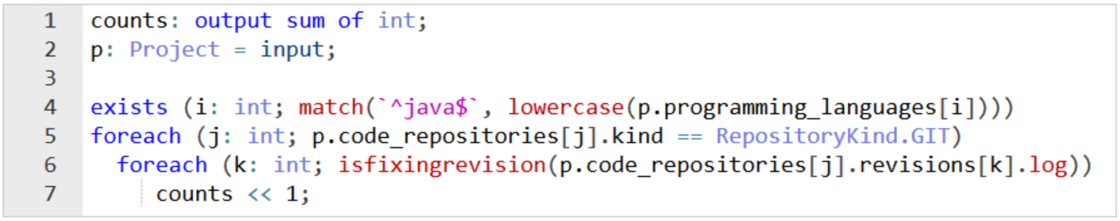
Fig. 3. Number of programming languages in each Java project using GIT.

IV. Results

In this section, we used the described dataset to answer the four research questions listed in the paper's introduction.

RQ$_1$: *Considering bug fixing commits associated with only one file, which file types are usually changed to fix a bug?*

Figure 4 shows the number of 1-file fixing commits per File Kind. As shown in Figure 4, the 2 kinds of changed files that appear most frequently in the 1-file fixing commits are: SOURCE_JAVA_JLS3 and UNKNOWN. The number of 1-file fixing commits related to these 2 kinds of changed files are respectively, 776,834 and 857,738. Text and binary files are changed least frequently. This is unsurprising, since such files are often documentation, and binaries should be changed rarely. XML files in Java projects usually represent build files or configuration files (the names of the most found configuration files end with "xml" or "properties" [11]); 7.7363% of analyzed 1-file fixing commits are related to changes in XML files. As these bugs are not related to source files, they could not be fixed by current automatic program repair techniques [11]. Rather more surprising is how frequently UNKNOWN files are changed. We deepen our analysis in these committed UNKNOWN files and found that


```
counts: output sum of int;
p: Project = input;

exists (i: int; match(`^java$`, lowercase(p.programming_languages[i])))
foreach (j: int; p.code_repositories[j].kind == RepositoryKind.GIT)
  foreach (k: int; isfixingrevision(p.code_repositories[j].revisions[k].log))
    counts << 1;
```

Fig. 1. Querying number of bug-fixing commits in Java GitHub projects using *Boa* language.

TABLE I
KINDS OF CHANGED FILES PRESENT IN THE *Boa* DATASET (JLS: JAVA LANGUAGE SPECIFICATION).

File Kind	Total	Description
SOURCE_JAVA_JLS4	83,798	The file represents a Java source file that parsed without error as JLS4
TEXT	541,023	The file represents a text file
BINARY	752,945	The file represents a binary file
SOURCE_JAVA_ERROR	2,073,558	The file represents a Java source file that had a parse error
SOURCE_JAVA_JLS2	2,607,413	The file represents a Java source file that parsed without error as JLS2
XML	6,818,299	The file represents an XML file
SOURCE_JAVA_JLS3	15,748,967	The file represents a Java source file that parsed without error as JLS3
UNKNOWN	23,426,568	The file's type was unknown

they are related to other programming languages like: C++, C, PHP, JavaScript, CoffeeScript, Erlang, Groovy, Scala, Python, Emacs Lisp, etc. Although the analyzed projects were mainly written in Java, 45,057 out of 101,471 Java projects (i.e., 44.4038%) use 2 or more programming languages (see Figure 3 for additional details). This results showed that 42.3352% of fixing commits that have changed 1 file are related to changes in non-Java source files.

Summary of $\mathbf{RQ_1}$. We notice that many bugs reside in non-Java source files (e.g., source files of different programming languages like Scala, Groovy, PHP, etc.) or non-source files (e.g., XML files). Many implementations of research techniques that automatically repair software bugs target programs written in C language (e.g., Prophet [5], GenProg [4]). Thus existing approach may be insufficient in fixing certain bugs. However, it is desirable to understand where such bugs reside, so we could investigate their nature and explore corresponding repair approaches.

$\mathbf{RQ_2}$: *Which kinds of statements appear more frequently in bug fixing commits?*

To answer this question, we use *Boa* to compute the number of fixing commits that added a particular statement kind in order to solve the corresponding bug. We investigate the following 13 statement kinds present in *Boa* Programming Guide [1]: `ASSERT`, `BLOCK`, `BREAK`, `CATCH`, `CONTINUE`, `FOR`, `IF`, `RETURN`, `SYNCHRONIZED`, `THROW`, `TRY`, `SWITCH`, and `WHILE`. The statement kind BLOCK is somewhat different because it was designed by *Boa* inventors to characterize a statement that contains a list of statements within it (e.g., the statements in the method body).

[1]http://boa.cs.iastate.edu/docs/dsl-types.php (verified on 20/07/2016)

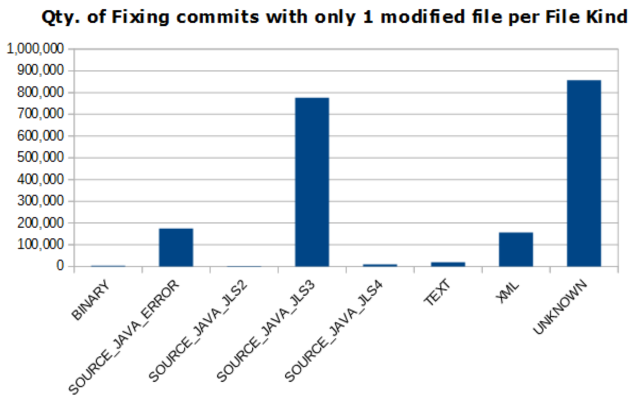


Fig. 4. Number of 1-file fixing commits per File Kind.

Concerning the `IF` Statement, we investigate how many fixing commits added null checks. The `IFNullCheck` statement is an `IF` statement where the boolean condition is of the form: `null == expr` OR `expr == null` OR `null != expr` OR `expr != null`. Basically, we build a query written in *Boa* language that counts how many null checks were previously in the file (previous version of the file, if exists) and how many null checks are currently in the file (actual version of the file). If there are more null checks than previously, the bug fixing commit is counted. We performed this algorithm for all changed files and fixing commits of our dataset (i.e., 52,052,571 and 4,590,405, respectively) and for all 13 statement kinds aforementioned.

Summary of $\mathbf{RQ_2}$. This research question is very important to investigate the nature of bug fixes in terms of statement kind that is added to fix a particular bug.

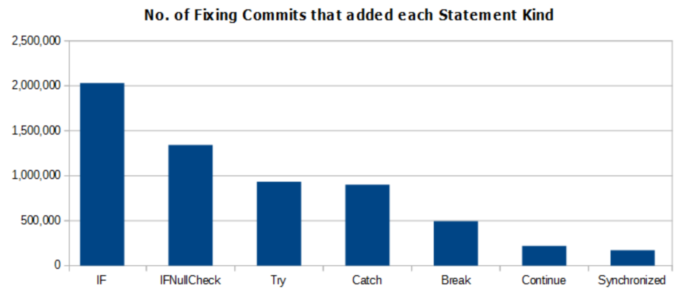


Fig. 5. Number of Fixing Commits that added each Statement Kind.

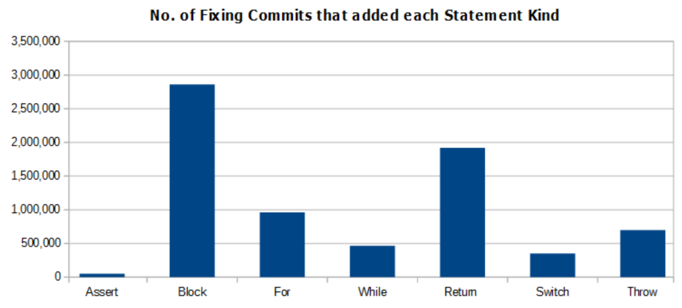


Fig. 6. Number of Fixing Commits that added each Statement Kind.

Moreover, we can identify the prevalence of some statement kinds with respect to others. For instance, Figures 5 and 6 show the results we obtained. Figure 5 shows the prevalence of `IF` and `IFNullCheck` statements with respect to the others (more specifically, 1,340,488 fixing commits added `IF` null checks). This number corresponds to 29% of all analyzed fixing commits. Moreover, Figure 6 shows the prevalence of `BLOCK` and `RETURN` statements with respect to other statement kinds like iteration commands (e.g., `WHILE`, `FOR`).

RQ$_3$: *What is the prevalence of the 5 most common bug fix patterns identified in the work of Pan et al. in the analyzed bug fixing commits?*

Pan *et al.* [8] identified 27 bug fix patterns through manual inspection of the bug fix change history of seven open-source Java projects. They found that the most common categories of bug fix patterns are Method Call and IF-related. Moreover, the most common individual patterns are: MC-DAP (Method Call with Different Actual Parameter Values), IF-CC (Change in IF conditional), and AS-CE (Change of Assignment Expression). Below we detail each one of these five bug fix patterns.

1) **Change of IF Condition Expression (IF-CC)**: The bug fix changes the condition expression of an `IF` condition [8]. Example:

```
- if (listBox.getSelectedIndex() == 0)
+ if (listBox.getSelectedIndex() > 0)
```

2) **Method Call with different actual parameter values (MC-DAP)**: The bug fix changes the expression passed into one or more parameters of a method call [8]. Example:

```
- String.getBytes("UTF-8");
+ String.getBytes("ISO-8859-1");
```

3) **Method Call with different number of parameters or different types of parameters (MC-DNP)**: The bug fix changes a method call by using different number of parameters, or different parameter types. This may be caused by a change of method interface, or use of an overloaded method [8]. Example:

```
- getSolrQuery(f.getFilter());
+ getSolrQuery(f.getFilter(),analyzer);
```

4) **Change of Assignment Expression (AS-CE)**: The bug fix changes the expression on the right hand side of an assignment statement. The expression on the left-hand side is the same in both the bug and fix versions [8]. Example:

```
- names[0] = person.getName();
+ names[0] = employees[0].getName();
```

5) **Addition of IF Precondition Check (IF-APC)**: This bug fix adds an `IF` predicate to ensure a precondition is met before an object is accessed or an operation is performed. Without the precondition check, there may be a `NullPointerException` error or an invalid operation execution caused by the buggy code [8]. Example:

```
- repo.getFileContent(path);
+ if (repo != null && path != null)
+     repo.getFileContent(path);
```

Pan *et al.* [8] also discovered that there is a similarity of bug fix patterns across projects. This indicates that developers may have trouble with individual code situations, and that frequencies of bug introduction are independent of application domain [8]. However, the main drawback of the bug fix patterns approach stems from its automation. We therefore automatically detect these five bug fix patterns, estimating their prevalence in the dataset presented in Section III.

We use *Boa* language to detect common bug fix patterns in the historical information of the projects. *Boa* provides domain-specific language features for mining source code [1]. *Boa's* capabilities are powerful, but limited in the precision it enables in detection of the aforementioned bug fix patterns. For example, it cannot directly `diff` two files. Rather than finding exact counts of bug fix patterns, we approximate by processing pre- and post-fix files separately. Fortunately, these five patterns can be detected by *Boa*, as we describe below. For each pattern, we create a query written in *Boa* language. In the following paragraphs, we describe in natural language each of the five algorithms designed to detect the five bug fix patterns aforementioned.

1) **How many bug fixing commits change one or more IF Condition Expressions (IF-CC)?** To answer this question and to detect this pattern, for both pre- and post-fix versions of a buggy file, we count how many `IF` conditions and expressions of these `IF` conditions appear. Then, if the number of `IF` conditions is the same between these two versions of the file (to ensure that it

is a modification and not an addition or deletion), we check whether the number of expressions of these `IF` conditions is different between these two versions of the file. If it's true, the pattern was detected and the bug fixing commit is recorded. For more information of what kind of expressions we consider, see the section `ExpressionKind` of this page [2]. *We found that 196,283 out of 4,590,405 (4.2759%) bug fixing commits change one or more* `IF` *condition expressions.*

2) **How many bug fixing commits change the parameter values of the method calls (MC-DAP)?** To answer this question and to detect this pattern, for both pre- and post-fix versions of a buggy file, we count how many method calls appear and we also built 2 strings (i.e., one string for the pre-version and another string for the post-fix version of these file) containing the parameter values (i.e., string literals) of all method calls. Then, if the number of method calls is the same between these two versions of the file, we compare if the two strings are different. If it's true, the pattern was detected and the bug fixing commit is recorded. *We found that 290,818 out of 4,590,405 (6.3353%) bug fixing commits change the parameter values of the method calls.*
3) **How many bug fixing commits change the number or type of parameters of the method calls (MC-DNP)?** To answer this question and to detect this pattern, for both pre- and post-fix versions of a buggy file, we count how many method calls and method parameters appear. Then, if the number of method calls is the same between these two versions of the file, we check whether the number of parameters are different. If it's true, the pattern was detected and the bug fixing commit is recorded. For this pattern, due *Boa* limitations, it was not possible to identify the types of method parameters present in the method calls. *We found that 192,375 out of 4,590,405 (4.1908%) bug fixing commits change the number of parameters of the method calls.*
4) **How many bug fixing commits change one or more assignment expressions (AS-CE)?** To answer this question and to detect this pattern, for both pre- and post-fix versions of a buggy file, we count how many assignment expressions and expressions of these assignments appear. Then, if the number of assignment expressions is the same (to ensure that it is a modification and not an addition or deletion), we check whether the number of expressions between these two versions of the file are different. If it's true, the pattern was detected and the bug fixing commit is recorded. For more information of what kind of expressions we consider, see the section `ExpressionKind` of this page [3]. *We found that 511,299 out of 4,590,405 (11.1384%) bug fixing commits change one or more assignment expressions.*
5) **How many bug fixing commits added a null check precondition (IF-APC)?** To answer this question and detect this pattern, for both pre- and post-fix versions of a buggy file, we count how many null checks appear. Then, if the number of null checks in the current version of the file is greater than in the previous version of these file, the pattern was detected and the bug fixing commit is recorded. The `IFNullCheck` statement is an `IF` statement where the boolean condition is of the form: `null == expr` OR `expr == null` OR `null != expr` OR `expr != null`. *We found that 1,340,488 out of 4,590,405 (29.2019%) bug fixing commits added a null check precondition.*

[2]http://boa.cs.iastate.edu/docs/dsl-types.php
[3]http://boa.cs.iastate.edu/docs/dsl-types.php

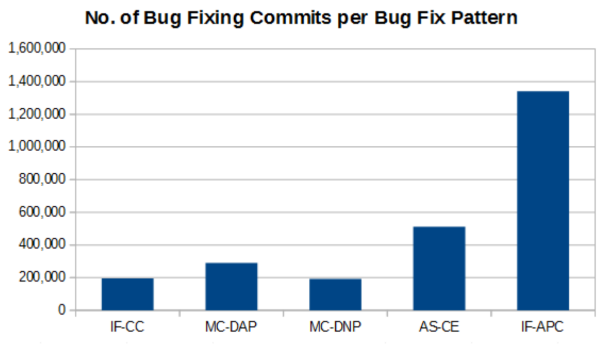


Fig. 7. Number of bug fixing commits per bug fix pattern.

Summary of RQ$_3$. Figure 7 shows a bar chart with the number of bug fixing commits distributed among the five studied bug fix patterns. As shown in this figure, the bug fix pattern that appears more frequently is IF-APC (29.2019% of the analyzed bug fixing commits). Observe that several bug fixing commits match this bug pattern in order to avoid *NullPointerException* errors.

RQ$_4$: How many distinct developers committed these bug fixing commits? And how many committers are there for each analyzed Java project?

To answer this research question, we build a *Boa* program that retrieves the username of the person who committed the revision. As the username is unique per person, it was possible to identify how many bug fixing commits each person committed and how many distinct developers committed these bug fixing commits. Figure 8 shows that a large number of open-source projects have only a single committer (33.6933%). Generally, open-source projects are small and have very few committers and thus problems affecting large development teams may not show when analyzing open-source software.

Summary of RQ$_4$: We found that only 130,488 distinct developers distributed among 101,471 Java projects committed 4,590,405 bug fixing commits. This shows how much the bug fixing commits are concentrated in a few people on the project and the need to better share knowledge in

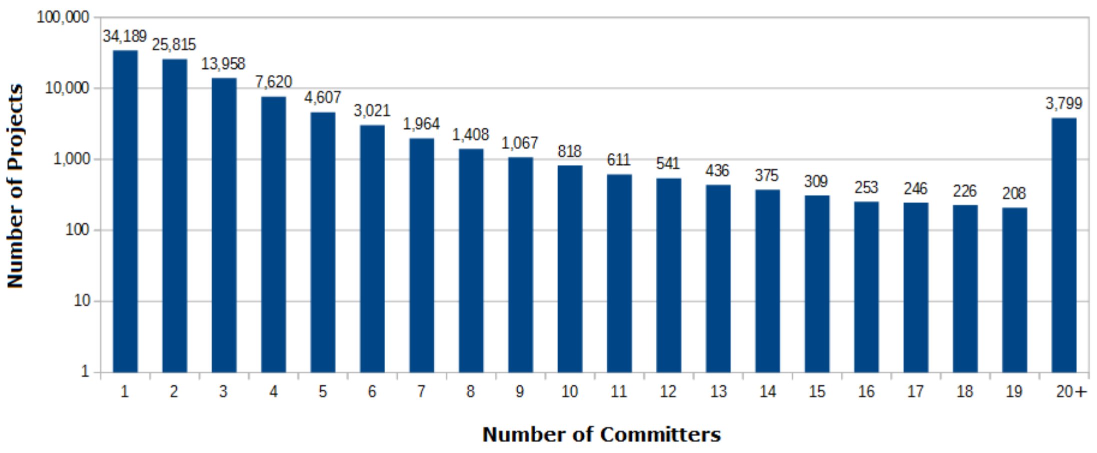


Fig. 8. Number of committers in each Java project using GIT.

software teams for future software maintenance tasks.

V. Threats to Validity

Correctness of Boa programs. The correctness of our analysis depends on both our *Boa* programs and its Domain-Specific Types [4]. For example, we rely on *Boa* to identify bug fixing commits. However, precisely accomplishing this is an open problem. To mitigate the risk of implementation errors, we released our *Boa* programs [5]. Because *Boa* does not provide an easy mechanism to identify precise, statement-level diffs between commits, our template matching and analysis of code changes (by counting each statement kind or expression kind) only provide estimates of behavior. We consider our results as informative approximations.

Systems are all open-source. All systems examined in this paper are developed as open-source. Hence they might not be representative of closed source development since different development processes could lead to different bug fix patterns. Despite being open-source, several of the analyzed projects have substantial industrial participation.

VI. Conclusion

This paper explored the underlying patterns in bug fixes mined from software project change histories. We rely on *Boa* to identify bug fixing commits and to detect the five most common bug fixing patterns identified by Pan *et al.* [8]. We have conducted a large-scale empirical study with more than 4 million bug fixing commits distributed among 101,471 Java projects hosted on GitHub to answer four research questions about bug fixes.

The findings of our study provide useful insights for automatic repair tools in Java. For example, future work may explore how to locate bugs in non-source files or source files of different programming languages present in a Java project and how to fix them with advanced techniques. Another example of follow-up work would be to propose an approach to automatic repair of assignment expressions (AS-CE bug fix pattern). Our work also confirms the results of a recent work [6] that showed that `IF` conditions are among the most error-prone program elements in Java programs (IF-APC bug fix pattern).

Overall, our findings motivate additional study of repair in Java and future research may leverage such knowledge to fix more bugs.

Acknowledgment

This work was partially supported by CAPES, CNPQ and FAPEMIG.

References

[1] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: Ultra-Large-Scale Software Repository and Source-Code Mining. *ACM Trans. Softw. Eng. Methodol.*, 25(1):7:1–7:34, Dec. 2015.

[2] D. Kim, J. Nam, J. Song, and S. Kim. Automatic Patch Generation Learned from Human-written Patches. In *Proc. of the ICSE '13*, pages 802–811, Piscataway, NJ, USA, 2013. IEEE Press.

[3] S. Kim and E. J. Whitehead, Jr. How Long Did It Take to Fix Bugs? In *Proc. of MSR '06*, pages 173–174. ACM, 2006.

[4] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *Proc. of ICSE '12*, pages 3–13, Piscataway, NJ, USA, 2012. IEEE Press.

[5] F. Long and M. Rinard. Automatic Patch Generation by Learning Correct Code. In *POPL '16*, pages 298–312, New York, NY, USA, 2016. ACM.

[6] M. Martinez and M. Monperrus. Mining Software Repair Models for Reasoning on the Search Space of Automated Program Fixing. *Empirical Software Engineering*, 20(1):176–205, Feb. 2015.

[7] M. Monperrus. A Critical Review of "Automatic Patch Generation Learned from Human-written Patches": Essay on the Problem Statement and the Evaluation of Automatic Software Repair. In *Proceedings of the ICSE' 2014*, pages 234–242, New York, NY, USA, 2014. ACM.

[8] K. Pan, S. Kim, and E. J. Whitehead, Jr. Toward an Understanding of Bug Fix Patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.

[9] M. Soto, F. Thung, C.-P. Wong, C. Le Goues, and D. Lo. A Deeper Look into Bug Fixes: Patterns, Replacements, Deletions, and Additions. In *Proc. of MSR '16*, pages 512–515. ACM, 2016.

[10] J. Xuan, M. Martinez, F. DeMarco, M. Clément, S. Lamelas, T. Durieux, D. Le Berre, and M. Monperrus. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering*, 2016.

[11] H. Zhong and Z. Su. An Empirical Study on Real Bug Fixes. In *Proc. of ICSE '15*, pages 913–923, Piscataway, NJ, USA, 2015. IEEE Press.

[4]http://boa.cs.iastate.edu/docs/dsl-types.php#Expression

[5]https://github.com/eduardocunha11/BoaPrograms (verified on 15/09/2016)