How Often Do Single-Statement Bugs Occur?
The ManySStuBs4J Dataset

Rafael-Michael Karampatsis
University of Edinburgh
Edinburgh, United Kingdom

Charles Sutton
Google Research, University of Edinburgh and The Alan
Turing Institute
Mountain View, CA, United States

ABSTRACT

Program repair is an important but difficult software engineering problem. One way to achieve acceptable performance is to focus on classes of simple bugs, such as bugs with single statement fixes, or that match a small set of bug templates. However, it is very difficult to estimate the recall of repair techniques for simple bugs, as there are no datasets about how often the associated bugs occur in code. To fill this gap, we provide a dataset of 153,652 single statement bug-fix changes mined from 1,000 popular open-source Java projects, annotated by whether they match any of a set of 16 bug templates, inspired by state-of-the-art program repair techniques. In an initial analysis, we find that about 33% of the simple bug fixes match the templates, indicating that a remarkable number of single-statement bugs can be repaired with a relatively small set of templates. Further, we find that template fitting bugs appear with a frequency of about one bug per 1,600-2,500 lines of code (as measured by the size of the project's latest version). We hope that the dataset will prove a resource for both future work in program repair and studies in empirical software engineering.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

ACM Reference Format:
Rafael-Michael Karampatsis and Charles Sutton. 2020. How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset . In *17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3379597.3387491

1 INTRODUCTION

Fixing bugs in programs, that is, program repair, is one of the core tasks in software maintenance, but requires effort to analyze failed executions, locate the cause of the fault, synthesize a bug fix and validate that the fault has been corrected without introducing new ones [19]. Automatic program repair [14, 16, 18, 20] attempts to

alleviate most of the manual effort of locating and repairing faults. However, a major concern in industry is that linters and program repair methods approaches are required to have high precision without risking achieving high enough recall. As an industrial example Google's Tricorder [22] enforces a false positive rate < 10%.

One way to find a "sweet spot" of maintaining high precision with adequate recall is to focus on repairing types of simple bugs, such as one-line bugs, or bugs that fall into a small set of templates, such as mutation operators [14] or other types of predefined templates [15, 16, 20]. However, these have been evaluated on either a relatively small numbers of projects, e.g. 69 defects in 8 applications or on synthetic data. Because of this lack of data, it has not previously been possible to estimate the *recall* of a set of repair templates, that is, the percentage of real-world bugs that can be repaired by one of the templates. Simultaneously to the current work, a larger dataset of one-line bugs has been mined [3], but even this dataset does not attempt to classify bugs into templates.

Aiming to fill this gap, we provide a dataset containing 25,539 single-statement bug-fix changes mined from 100 popular open-source Java Maven projects as well as a larger one containing 153,652 single-statement bug-fix changes mined from 1,000 popular open-source Java projects, annotated by whether they match any of a set of 16 bug templates, inspired by state-of-the-art program repair techniques. The chosen templates aim at extracting bugs that compile both before and after repair as such can be quite tedious to manually spot, yet their fixes are so simple that many developers would call them "stupid" upon realization. We will refer onwards to these bugs as "simple stupid bugs" (SStuBs)[1] and the corresponding dataset as the ManySStuBs4J dataset. Automatic repair of SStuBs is potentially an intermediate step toward more general program repair tools, while already being useful to developers. We also think that SStuBs might be a good start for the evaluation of machine learning based fault localization and repair methods.

An extra distinctive feature of our dataset is that the smaller version is restricted to projects that can be built automatically using Maven. Those that contain a test suite can be built and used to evaluate test based techniques. In an initial analysis, we find that 33.04% in the smaller version dataset and 33.47% in the larger version of all of the single-statement bugs that we mine match at least one of the SStuB templates resulting in 10,231 and 63,923 SStuB instances respectively. This indicates that a remarkable number of singe-statement bugs can be repaired with a relatively small set of templates. In further analysis we also estimated the frequency in lines of code with which these pattern based and general single-statement bugs appear. This estimation is based on the size of

---

[1]The acronym is intended to reflect the fact that, for the authors at least, finding such a bug can feel much like stubbing one's toe.

the project's latest version and reveals that in the smaller dataset version SStuBs appear with a frequency of about 1 per 1,600 lines of code and 1 per 2,500 lines of code for the large version. We hope that this dataset can serve as a valuable resource for both future work in program repair and studies in empirical software engineering.

2 METHODOLOGY

We next describe the methodology we employed to build the dataset. Our data generation tools along with documentation and detailed instructions for how to use them are available in a public GitHub repository[2] and the dataset is publicly available in Zenodo.[3]

2.1 Selecting Appropriate Java Projects

In order to mine a high quality dataset we opted to selecting high popularity projects. For the small version of the dataset we selected the 100 most popular open source Java Maven [17] projects from GitHub up to 1/4/2017. To allow evaluation of repair tools that might require building the projects, we selected only Maven ones because it is easy to automatically download the required dependencies for every project and build it. In contrast, manual downloading of dependencies would require an immense amount of human effort. To create a ranking for the projects we downloaded the MySQL dump of GHTorrent [7] up to 1/4/2017. A project's popularity is determined by computing the sum of z-scores of its forks and stars [1, 2]. Lastly, we pulled the projects' head commit by 28/1/2019 and considered commits until that date. The same approach was used to rank projects for the larger version. However, the ranking was calculated using a later dump of GHTorrent from 1/1/2019. A download script along with the list of projects for both variants of the dataset are also available to ensure replicability.

2.2 Classifying Commits as Bug-Fixing or not

For every project our tool searches historically through all of its commits to locate bug-fixing ones. To decide if a commit fixes a bug, we checked if its commit message contains at least one of the keywords: 'error', 'bug', 'fix', 'issue', 'mistake', 'incorrect', 'fault', 'defect', 'flaw', and 'type'. This heuristic was previously used by Ray et al. [21] and was shown to achieve 96% accuracy on a set of 300 manually verified commits and 97.6% on a set of 384 manually verified commits [25]. We sampled 100 random commits containing SStuBs from the small version of the dataset and found it to achieve 94% accuracy. The above process produced a total of 115,929 and 883,982 bug-fixing commits for the small and large dataset variants.

2.3 Selecting Single Statement Changes

We have opted to restrict the dataset to small bug fixes that do not require much code modification to fix. Additionally, we are interested in bugs that are not just syntactic errors but cases where the code compiles both before and after the bug was located and repaired. As we are interested in simple bugs that involve only a single statement, we filter out any commits that either add or delete a Java file. We also filter out commits which make a multiple-statement change at any single position in the Java file. We do *not* filter out commits that make single-line modifications at more than

one position in the same file. Similarly to the diff algorithm, we consider a modification as deleting the old lines/statements and then adding the new ones. To estimate whether a modification spans across multiple statements we calculate the diff for each modified Java file, and for each modified chunk, we count how many statements were modified. In the case of blocks each statement in the block's body is counted as a different statement. For `if` and `while` statements, we count the condition as a separate statement for this purpose. This method allows to us include fixes to single simple statements that span across multiple lines (e.g. due to stylistic reasons) as a simple fix, unlike a line-based approach. Any commits that modify multiple statements in any single position returned by the diff are dropped while we still maintain commits for which a file's diff contains multiple positions with single statement modifications. In the first case it is not trivial to align the deleted and added statements while it is in the latter. For example, one or more of the deleted statements may have been replaced by multiple of the added ones while simultaneously one or more of the deleted statements may have simply been deleted. We note that our tool ignores any changes to comments, blank lines as well as any formatting changes. Our methodology allows cases where the same expression containing a bug appeared multiple times in the file. This filtering produces almost 13,000 and 86,769 commits for the two dataset versions. Lastly, the employed methodology works in a similar way to the popular SZZ algorithm [24] and its extensions [13, 27] that have extensively been used to spot fix inducing changes.

2.4 Creating Abstract Syntax Trees

Each file in the commit that contains one or more bugs is parsed, yielding an abstract syntax tree (AST) of the file before the repair. Then, for each repaired line in the file we extract the AST after applying the repair only on that line and leaving the rest of the lines as is. Each extracted pair of ASTs (original and single fix) only differ on the node(s) for the modified line. By performing a simultaneous depth-first traversal on the two ASTs we locate the first node on which the two ASTs differ.

2.5 Filtering out Clear Refactorings

Although we filter for bug-fixing changes in Step B, there might still exist changes in the data that do not fix a bug or that do not even produce any behavioural changes. This could happen because the commit-message filter had a false positive, or because the change is tangled [10], and contains a bug-fixing modification along with unrelated ones to other files. To reduce the number of non-fixing changes in the dataset, we observe that there is a class of refactorings that can produce small changes, namely renamings. These are extracted via the diffs of the modified files. Our method spots variable, function, or class renaming as well as any uses of them across other modified files in the commit and excludes them.

2.6 SStuB Patterns

We next describe the 16 SStuB patterns. We opted to choose patterns that appear often. Many of these have been used in pattern-based repair and mutation tools [14–16, 20]. Here we provide a brief description of each pattern. Due to page limitations we do not include examples here but in the README of the GitHub repository.

- *Change Identifier Used* Checks whether an identifier appearing in some expression in the statement was replaced with another one. It is easy for developers to by accident utilize a different identifier than the intended one that has the same type. Copy pasting code is a potential source of such errors. Similarly named identifiers may further contribute to the occurrence of such errors.
- *Change Numeric Literal* Checks whether a numeric literal was replaced with another one. It is easy for developers to mix two numeric values in their program.
- *Change Boolean Literal* Checks whether a Boolean literal was replaced. True is replaced with False and vice-versa. In many cases developers use the opposite Boolean value than the intended one.
- *Change Modifier* Checks whether a variable, function, or class was declared with the wrong modifiers. For example a developer can forget to declare one of the modifiers.
- *Wrong Function Name* Checks if a function with the same parameter list but the wrong name was called. This is a usual pitfall.
- *Same Function More Args* Checks whether an overloaded version of the function with more arguments was called. Functions with multiple overload can often confuse developers.
- *Same Function Less Args* Checks whether an overloaded version of the function with less arguments was called. For instance, a developer can forget to specify one of the arguments and not realize it if the code still compiles due to function overloading.
- *Same Function Change Caller* Checks whether in a function call expression the caller object for it was replaced with another one. When there are multiple variables with the same type a developer can accidentally perform an operation. Copy pasting code or mixing similar variables are common cases of such errors.
- *Same Function Swap Args* Checks whether a function was called with two of its arguments swapped. When multiple function arguments are of the same type, developers can easily swap two of them without realizing. It was also used in DeepBugs [20].
- *Change Binary Operator* Checks whether a binary operand was accidentally replaced with another one of the same type. For example, developers very often mix comparison operators in expressions. A similar pattern was also used in DeepBugs [20].
- *Change Unary Operator* Checks whether a unary operand was accidentally replaced with another one of the same type (e.g., developers often forget the ! operator in a boolean expression).
- *Change Operand* Checks whether one of the operands in a binary operation was wrong. It was also used in DeepBugs [20].
- *More Specific If* Checks whether an extra condition (&& operand) was added in an `if` statement's condition.
- *Less Specific If* Checks whether an extra condition which either this or the original one needs to hold (|| operand) was added in an `if` statement's condition.
- *Missing Throws Exception* Checks whether the fix added a `throws` clause in a function declaration.
- *Delete Throws Exception* Checks whether the fix deleted a `throws` clause in a function declaration.

2.7 SStuB Pattern Matching

Finally, each pair of ASTs is automatically checked for fitting any of the SStuB patterns. Each pattern is expressed as a mutation operation on the original AST that produces the new one. All instances

Table 1: Statistics for each SStuB pattern.

Pattern Name	SStuBs	Ratio	SStuBs L	Ratio L
Change Identifier Used	3265	12.78%	22668	14.75%
Change Numeric Literal	1137	4.45%	5447	3.55%
Change Modifier	1852	7.25%	5011	3.26%
Change Boolean Literal	169	0.66%	1842	1.20%
Wrong Function Name	1486	5.82%	10179	6.62%
Same Function More Args	758	2.97%	5100	3.32%
Same Function Less Args	179	0.70%	1588	1.03%
Same Function Wrong Caller	187	0.73%	1504	0.98%
Same Function Swap Args	127	0.50%	612	0.39%
Change Binary Operator	275	1.08%	2241	1.46%
Change Unary Operator	170	0.67%	1016	0.66%
Change Operand	120	0.47%	807	0.53%
Less Specific If	215	0.84%	2813	1.83%
More Specific If	175	0.69%	2381	1.55%
Missing Throws Exception	68	0.27%	206	0.13%
Delete Throws Exception	48	0.19%	508	0.33%
TOTAL NO DOUBLE COUNTS	8438	33.04%	51433	33.47%
TOTAL	10231	40.06%	63923	41.59%

are added to the single-statement dataset, while only those that match SStuB patterns are saved in the SStuBs one.

3 MANYSSTUBS4J DATASET STATISTICS

The ManySStuBs4J dataset consists of 10,231 and 63,923 instances of single statement bugs mined from 12,598 and 86,771 bug-fix commits with only single-statement changes respectively for each version. Consequently, on average almost 2 single statement bugs and 0.75 SStuBs were mined per valid commit. The data is saved in JSON files and detailed information is available in the GitHub repository. Each SStuB instance is also annotated with the SStuB pattern satisfied, the project's name, the Java file's name, the hashes of the fix inducing commit and its parent, the line at which the bug starts, and the AST subtree's location. In some cases a statement might fit more than one patterns. In those cases it is counted as separate instances. However, in most cases the patterns are distinct. The statistics for each of the 16 SStuB patterns of the ManySStuBs4J dataset are shown in Table 1. Patterns that are similar are grouped together (e.g. patterns that concern functions) and sorted in descending frequency order. The three most common SStuB patterns are *Change Identifier Used*, *Wrong Function Name*, and *Change Numeric Literal*.

We note that the mined bugs have not been annotated by severity and we expect that to vary. Some of the bugs appear in test code. Although bugs in test code will not reach a final product, they can have significant effect on it as they can potentially mask important bugs in it. Test oracle errors can bring confusion that slows down the debugging process while fixing them improves the performance of fault localization algorithms [8]. Such bugs might also be quite tedious to locate as it is very rare to test a test suite and even if we follow that logic we would have to endlessly create tests for the tests. By design we do not attempt to restrict the bugs to those that have a failing test case. The goal is to reproduce the situations that the bugs happen in the wild. Lastly, as it was recently shown, unit tested code does not appear to be associated with fewer failures while increased coverage is associated with more failures [5].

4 RESEARCH QUESTIONS

Although the paper focuses on the dataset, we run a simple analysis to support our design decision to focus our new dataset on SStuBs. In order to explore whether the SStuB patterns are useful targets for program repair techniques, we asked two research questions.

RQ1. Are SStuBs common in open-source code?

We measured for each SStuB type the percentage of single statement modifications that are not clear refactorings and fit the pattern. These are visualized in Table 1. For each project P we also estimated the following two densities for the mined SStuBs: (a) the number of SStuBs in project P / total lines in P at the final snapshot and (b) the number of SStuBs in project P / total lines added and deleted in P by the final snapshot. Thus, estimating the frequency per line of code modifications in the project's history. That is counting any line that was added or deleted to the project from the start to its latest version. A line modification is counted twice (once as a deletion and once as an addition). Once for deleting the old and once for adding the new line. Comments and empty lines were excluded from these estimations. We found that in the smaller version of the dataset SStuBs appear with densities of about 2,400 and 30,000 lines of code (LOC) respectively.

We also estimated the same densities for the larger dataset variant. We found that such bugs appear with a frequency of about 1,600 and 20,000 LOC respectively. As a threat to validity, we acknowledge that the number of LOC in the final snapshot may not be the most informative denominator for a measure of bug density, but developing better ones is a thorny issue left for future work.

RQ2. Can SStuBs be spotted by existing tools such as static analyzers?

We measure the proportion of bugs in our dataset that can be identified by the popular static analysis tool SpotBugs.[4] If SpotBugs reports any bug for the line containing the SStuB then we consider that SpotBugs successfully detected it. We find that SpotBugs could only locate about 12% of SStuBs while also reporting more than 200 million possible bugs when configured to report all warnings, even those with low confidence. In fact, as explained the actual recall is even lower. This is confirmed by a recent study where three static bug detectors including SpotBugs located only 4.5% of bugs [9]. This means that a developer would have to look through hundreds of thousands of warnings produced by SpotBugs to locate a single SStuB. This highlights the necessity for tools that are specifically built to detect SStuBs. The scripts used to run and evaluate SpotBugs are also available in our repository.

5 RELATED WORK

Several previous data sets of real-world bugs have been curated. Defects4J [11] is a popular dataset consisting 395 Java bugs. Each bug is fixed in a single commit but the fix may modify multiple source code lines. The ManyBugs dataset [6] contains 185 C bugs, a subset of which were used by the GenProg [14], Prophet [16] and SPR [15] papers. Bugs.jar [23] is comprised of 1,158 Java bugs and their patches. These datasets have the disadvantage of being relatively

small. More recently, a few larger-scale data sets of small bugs have been created. The combined datasets are the CodRep dataset [4] and the Bugs2Fix dataset [26] resulting in 40,289 one-line bugs. These datasets are combined into a single dataset of one line bugs in [3]. Our datasets are of similar size consisting of 25,539 and 153,652 single-statement bugs. In contrast, our dataset focus on estimating the frequency of SStuB templates, motivated by recent program repair tools and also operates on the statement level, which prevents falsely excluding instances due to formatting or stylistic reasons. Also, the projects from which the small version of our dataset was generated can easily be built using Maven and we provide a list of projects containing tests and which tests fail for each instance (in GitHub repo). Thus test based methods can be evaluated upon them. However, unlike Defects4J that aims in comparing test-based patch generation approaches, it aims in techniques that can accurately highlight SStuBs early in development allowing immediate patching since in many cases the fix might be trivial. Lastly, unlike previous datasets, we take additional steps to filter out refactorings, although we acknowledge that such instances might be rare. In our case however, we were able to filter out almost 5,000 and 35,000 refactored statements for the two dataset versions.

6 LIMITATIONS - THREATS TO VALIDITY

Although unlikely, it is possible for our SZZ like methodology to extract a pair of aligned statements that are unrelated (i.e. one line was deleted and one was added). We do spot refactorings but there is no guarantee that we have detected 100% of them. The heuristic used to spot bug fixing commits could introduce false positives, but this is mitigated by the fact that we focus on single line commits and as already discussed the false positive rate is low. Our dataset will not be useful for evaluating whether repair systems are good at fixing larger bugs. Our dataset is restricted to Java but could be replicated for other languages by using a parser and creating a module that checks if an AST pair fits any of the SStuB patterns. The precise set of patterns might vary across languages and determining these might be an interesting direction for future work.

7 CONCLUSIONS

We introduce a new, large-scale dataset of real-world SStuBs, simple one-statement bugs, in Java for the evaluation of program repair techniques. The distinguishing feature of our dataset is that where possible, the SStuBs are categorized into one of 16 bug templates, which are inspired by those considered in state-of-the-art program repair methods. These types of bugs often result in code that compiles, which means that they are particularly interesting for automated repair. We find that SStuBs occur relatively often — one per 1,600 LOC in the projects we study — making them potentially a promising evaluation dataset for repair techniques that could be used to estimate their actual recall. The data could also be used to answer other research questions, such as empirical questions about how and when simple bugs are introduced, or about evaluating program repair techniques for small bugs. Also, it can aid in evaluating machine learning systems that learn to localize simple bugs via examples [20] or a language model's entropy [12]. Last, coverage information for the maven projects with tests suits in the dataset could be used to estimate how often do tests cover SStuBs.

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting Accurate Method and Class Names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. ACM, New York, NY, USA, 38–49. https://doi.org/10.1145/2786805.2786849

[2] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *Proceedings of ICML 2016*, Vol. 48. 2091–2100. http://proceedings.mlr.press/v48/allamanis16.html

[3] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *CoRR* abs/1901.01808 (2019). arXiv:1901.01808 http://arxiv.org/abs/1901.01808

[4] Zimin Chen and Martin Monperrus. 2018. The CodRep Machine Learning on Source Code Competition. *CoRR* abs/1807.03200 (2018). arXiv:1807.03200 http://arxiv.org/abs/1807.03200

[5] Efstathia Chioteli, Ioannis Batas, and Diomidis Spinellis. 2019. Does Unit-Tested Code Crash? A Case Study of Eclipse. *arXiv preprint arXiv:1903.04055* (2019).

[6] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering* 41, 12 (Dec 2015), 1236–1256. https://doi.org/10.1109/TSE.2015.2454513

[7] Georgios Gousios. 2013. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories* (San Francisco, CA, USA) *(MSR '13)*. IEEE Press, Piscataway, NJ, USA, 233–236. http://dl.acm.org/citation.cfm?id=2487085.2487132

[8] X. Guo, M. Zhou, X. Song, M. Gu, and J. Sun. 2015. First, Debug the Test Oracle. *IEEE Transactions on Software Engineering* 41, 10 (Oct 2015), 986–1000. https://doi.org/10.1109/TSE.2015.2425392

[9] Andrew Habib and Michael Pradel. 2018. How Many of All Bugs Do We Find? A Study of Static Bug Detectors. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 317âĂŞ328. https://doi.org/10.1145/3238147.3238213

[10] Kim Herzig and Andreas Zeller. 2013. The impact of tangled code changes. In *Working Conference on Mining Software Repositories*. IEEE Press, 121–130.

[11] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. ACM, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[12] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code. arXiv:cs.SE/2003.07914

[13] S. Kim, T. Zimmermann, K. Pan, and E. J. Jr. Whitehead. 2006. Automatic Identification of Bug-Introducing Changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. 81–90. https://doi.org/10.1109/ASE.2006.23

[14] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Software Eng.* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[15] Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. ACM, New York, NY, USA, 166–178. https://doi.org/10.1145/2786805.2786811

[16] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. *SIGPLAN Not.* 51, 1 (Jan. 2016), 298–312. https://doi.org/10.1145/2914770.2837617

[17] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. 2010. *Apache Maven*. Alpha Press.

[18] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1, Article 17 (Jan. 2018), 24 pages. https://doi.org/10.1145/3105906

[19] Monika A. F. Müllerburg. 1983. The Role of Debugging Within Software Engineering Environments. *SIGPLAN Not.* 18, 8 (March 1983), 81–90. https://doi.org/10.1145/1006142.1006165

[20] Michael Pradel and Koushik Sen. 2018. DeepBugs: A Learning Approach to Name-based Bug Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 147 (Oct. 2018), 25 pages. https://doi.org/10.1145/3276517

[21] Baishakhi Ray, Vincent Hellendoorn, Zhaopeng Tu, Connie Nguyen, Saheel Godhane, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the" Naturalness" of Buggy Code *(ICSE '16)*. ACM.

[22] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Soederberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *International Conference on Software Engineering (ICSE)*.

[23] Ripon K. Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R. Prasad. 2018. Bugs.Jar: A Large-scale, Diverse Dataset of Real-world Java Bugs. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. ACM, New York, NY, USA, 10–13. https://doi.org/10.1145/3196398.3196473

[24] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes?. In *International Workshop on Mining Software Repositories*. ACM.

[25] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. An Empirical Investigation into Learning Bug-fixing Patches in the Wild via Neural Machine Translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. ACM, New York, NY, USA, 832–837. https://doi.org/10.1145/3238147.3240732

[26] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *CoRR* abs/1812.08693 (2018). arXiv:1812.08693 http://arxiv.org/abs/1812.08693

[27] Chadd Williams and Jaime Spacco. 2008. SZZ Revisited: Verifying when Changes Induce Fixes. In *Proceedings of the 2008 Workshop on Defects in Large Software Systems* (Seattle, Washington) *(DEFECTS '08)*. ACM, New York, NY, USA, 32–36. https://doi.org/10.1145/1390817.1390826